Terraforming—Environment Manipulation during Disruptions for Multi-Agent Pickup and Delivery

David Vainshtein, Yaakov Sherma, Kiril Solovey, Oren Salzman *

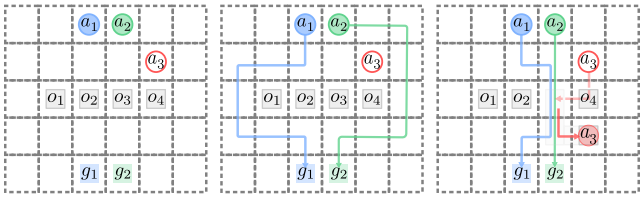
Technion - Israel Institute of Technology, Haifa, Israel
dudiwa@cs.technion.ac.il, yaakovsherma@cs.technion.ac.il, kirilsol@technion.ac.il, osalzman@cs.technion.ac.il

Abstract

In automated warehouses, teams of mobile robots fulfill the packaging process by transferring inventory pods to designated workstations while navigating narrow aisles formed by tightly packed pods. This problem is typically modelled as a Multi-Agent Pickup and Delivery (MAPD) problem, which is then solved by repeatedly planning collision-free paths for agents on a fixed graph, as in the Rolling-Horizon Collision Resolution (RHCR) algorithm. However, existing approaches make the limiting assumption that agents are only allowed to move pods that correspond to their current task, while considering the other pods as stationary obstacles (even though all pods are movable). This behavior can result in unnecessarily long paths which could otherwise be avoided by opening additional corridors via *pod manipulation*. To this end, we explore the implications of allowing agents the flexibility of dynamically relocating pods. We call this new problem Terraforming MAPD (tMAPD) and develop an RHCR-based approach to tackle it. As the extra flexibility of terraforming comes at a significant computational cost, we utilize this capability judiciously by identifying situations where it could make a significant impact on the solution quality. In particular, we invoke terraforming in response to *disruptions* that often occur in automated warehouses, e.g., when an item is dropped from a pod or when agents malfunction. Empirically, using our approach for tMAPD, where disruptions are modeled via a stochastic process, we improve throughput by over 10%, reduce the maximum *service time* (the difference between the drop-off time and the pickup time of a pod) by more than 50%, without drastically increasing the runtime, compared to the MAPD setting.

1 Introduction

Multi-Agent Path Finding (MAPF) is a popular algorithmic framework that captures complex tasks involving mobile agents that need to plan individual routes while avoiding collisions during plan execution (Stern et al. 2019; Salzman and Stern 2020). This abstraction has been successfully applied to a variety of settings (see, e.g., Wurman, D'Andrea, and Mountz (2008); Belov et al. (2020); Li et al. (2021a); Greshler et al. (2021); Choudhury et al. (2021)). However, in some cases this formulation may not be expressive enough to fully capture the underlying task, which can lead to suboptimal performance.



(a) MAPF problem (b) Static obstacles (c) Terraforming

Figure 1: Comparing MAPF and tMAPF for a toy problem with agents a_1, a_2, a_3 (circles), and a row of obstacles (grey squares). (a) MAPF problem assigning agents to their goal locations (squares). Here, agent a_3 is a *free* agent which does not carry a pod. (b) MAPF solution where agents must avoid collisions with obstacles and each other. Here, agent a_3 cannot help the other two agents as all obstacles are static. (c) tMAPF solution where a_3 displaces the movable obstacle o_3 to open a passage.

This is especially true in the context of automated warehouses where we are given a stream of tasks, and the goal is to maximize the system's throughput. In this setting, formulated as a Multi-Agent Pickup and Delivery (MAPD) problem, inventory pods that hold goods are transported by a large team of mobile agents: agents pick up pods, carry them to designated drop-off locations, where goods are manually removed from the pods (to be packaged for customers); each pod is then carried back by a robot to a (possibly different) storage location (Wurman, D'Andrea, and Mountz 2008). MAPD is typically solved via a sequence of MAPF queries, as in the RHCR algorithm and its successors (Ma et al. 2017; Liu et al. 2019; Li et al. 2021b; Madar, Solovey, and Salzman 2022). Existing variants of MAPF and MAPD tend to impose the following limiting and artificial constraint: pods that are not currently carried to or from a drop-off location are modeled as *static obstacles*, which cannot be moved. Such approaches overlook the fact that pods can be manipulated to clear the way for agents and reduce execution time.

*This research was supported in part by the Israeli Ministry of Science & Technology grants No. 3-16079 and 3-17385, the United States-Israel Binational Science Foundation (BSF) grants no. 2019703 and 2021643, the Amazon Research Award, and Ravitz Foundation.
Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

To this end, we explore the implications of allowing agents the extra flexibility of *manipulating the environment by moving obstacles* (i.e., dynamically relocating pod's locations in warehouse applications). Specifically, we introduce a new MAPD variant (depicted in Fig. 1) which we term *"Terraforming MAPD"* (tMAPD).[1] Unfortunately, the extra flexibility of environment manipulation comes at a computational price—solving a tMAPD problem takes significantly longer than a MAPD problem. Roughly speaking, this is because a planner needs to (i) consider which static obstacles to displace and (ii) which agent should perform said displacements. Thus, in this work we suggest to use terraforming only when unexpected *disruptions* are experienced.

In real-life settings disruptions may occur when an item is dropped from a pod or when agents experience malfunctions. Such events require a safety perimeter to be enforced, cutting off routes and requiring agents to replan in order to detour the disruption area. In the extreme case, agents may become completely trapped by surrounding disruptions, rendering them unable to carry out their tasks. As we will see, such events are excellent candidates for environment manipulation: A disruption can be used to guide a planner which obstacles to consider as candidates to be moved allowing the planner to heuristically focus its computation. More importantly, solutions computed by a planner that can manipulate the environment are often of much higher quality when compared to planners that do not have this extra flexibility making the additional computational effort worthwhile.

We develop an algorithmic approach to tackle the tMAPD problem with disruptions, which casts the question of which obstacles should be moved and by whom to MAPF-like problems. Those ideas, which are combined with an RHCR-based approach, which we call Terraforming RHCR (tRHCR), dramatically reduce the size of the huge decision space and allow to efficiently solve the tMAPD problem. We evaluate the benefit of tRHCR on warehouse environments that are affected by disruptions, empirically demonstrating an improvement in throughput by over 10% and a reduction of the maximum *service time* (the difference between the drop-off time and the pickup time of a pod) by more than 50%, without drastically increasing the runtime, as compared with a standard MAPD approach.

2 Related Work

Research on MAPF has produced a wide variety of approaches, ranging from network flow (Yu and LaValle 2012), to satisfiability (Surynek et al. 2016), answer set programming (Erdem et al. 2013) and tree-search methods (Barer et al. 2014; Boyarski et al. 2015; Sharon et al. 2015; Li et al. 2019). With the focus of this work on the latter group, the method of the most relevance to ours is Priority-Based Search (PBS) (Ma et al. 2019a) which offers a balance between solution quality and fast computation when solving the MAPD problem (Ma et al. 2017; Li et al. 2021b). In our work we utilize PBS (detailed in Sec. 4) both as our baseline MAPF solver and as the basis for our tMAPD algorithm.

The most closely-related work to our new terraforming problem formulation is by Bellusci, Basilico, and Amigoni (2020), in which a configurable environment is optimized alongside path-finding efforts of MAPF. Referred to as the Configurable MAPF (C-MAPF) problem, it allows for the reconfiguration of the environment, subject to constraints imposed on the graph itself. An important distinction from our work is that solving the C-MAPF problem consists of searching for a *fixed* graph resulting in minimal graph alterations, as well as a set of valid paths dictating where each agent should go. In contrast, the environment in tMAPD has the capacity to *dynamically change* as agents execute their paths and obstacles are temporarily cleared to make way.

Also related to our work is a generalization of MAPF called *k-robust* MAPF (Atzmon et al. 2018) that produces paths guaranteed to be collision-free even when agents are delayed by up to k timesteps. The concept of robustness is demonstrated as an effective mechanism for avoiding collisions in the context of sudden delays that occur with probability p per each move of each agent (Atzmon et al. 2020). In our work, although the probability of future per-agent delays p can be modeled, we account for disruptions that block sections of the graph not occupied by an agent and their duration is not known in advance.

Finally, we mention that the state-of-the-art approach for MAPD is the Rolling-Horizon Collision Resolution (RHCR) algorithm (Li et al. 2021b). RHCR iteratively plans a set of partial paths for a group of agents up to a specified time-horizon, decomposing MAPD into a series of MAPF queries that are solved iteratively. The choice of MAPF solver is very often PBS as it offers a balance of solution quality and fast computation time. In our work we extend RHCR to incorporate terraforming and to account for unexpected disruptions that affect the graph itself.

3 Problem Formulation

In this section we start by defining the MAPF problem and the continue to define the Multi-Agent Pickup and Delivery (MAPD) problem and introduce the notion of disruptions in the context of MAPD. Finally, we define the tMAPD problem, which is used in our approach to efficiently handle disruptions within MAPD settings.

3.1 Multi-Agent Path Finding (MAPF)

We define the MAPF problem as a tuple $\langle \mathcal{G}, \mathcal{A}, \mathcal{V}_{\text{start}}, \mathcal{V}_{\text{goal}}, \mathcal{O} \rangle$, where the environment $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph and $\mathcal{A} = \{a_1, \ldots, a_n\}$ is the set of agents. Here, $\mathcal{V}_{\text{start}} = \{s_1, \ldots, s_n\}$ and $\mathcal{V}_{\text{goal}} = \{g_1, \ldots, g_n\}$ are the agents' start and goal vertices, respectively. Agents move between vertices along graph edges and are allowed to wait in place. For each agent a_i, its actions occur in discrete timesteps of unit cost, resulting in a path π_i comprised of a sequence of vertices $\pi_i = \langle s_i, \ldots, g_i \rangle$ that is associated with a cost $|\pi_i|$ of the total number of actions.

A solution to the MAPF problem is a set of collision-free paths $\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_n\}$ such that no two agents share

[1] The concept of "Terraforming" emerged in science fiction at the dawn of the space race, as a means of space exploration, in which a planet's inhospitable environment is altered to facilitate life. For additional details, see, e.g., https://sfdictionary.com/view/125/terraforming.

the same vertex at the same timestep $\pi_i[t] \neq \pi_j[t]$, nor are they allowed to cross over the same edge in opposing directions $(\pi_i[t], \pi_i[t+1]) \neq (\pi_j[t+1], \pi_j[t])$. The cost of the solution $\boldsymbol{\pi}$ is called its *flowtime* and is defined as the sum of individual path costs. Namely, $|\boldsymbol{\pi}| = \sum_i |\pi_i|$ with $|\pi_i|$ denoting the cost (number of timesteps) of the solution of agent i. Note that other cost functions exist, such as *makespan* (Stern et al. 2019) that corresponds to the maximum path cost among all agents, i.e., $\max\{|\pi_i|\}_i$.

In typical MAPF formulations, *static obstacles* that block certain agent positions are implicitly represented by omitting blocked vertices and edges from the graph $\mathcal{G}$. In our setting however, we facilitate interactions between agents and obstacles by explicitly denoting vertices that the agents cannot visit as a set of obstacles $\mathcal{O} = \{o_1, \ldots, o_n\} \subset \mathcal{V}$.

3.2 Multi-Agent Pickup and Delivery (MAPD)

We now describe the standard setting of MAPD where we are tasked with continuously planning for agents as they handle tasks assigned from a task queue $\mathcal{T}$. In this work we assume for simplicity that tasks arrive as an online stream, meaning we do not have access to future tasks.

Each task $\tau_i \in \mathcal{T}$ consists of a pair $\langle p_i, d_i \rangle$ where $p_i \in \mathcal{O}, d_i \in \mathcal{V}$ are its pickup and delivery locations, respectively. When an agent is assigned a task, it must (i) arrive at the obstacle's pickup location p_i, (ii) carry and deliver it to the delivery location d_i and (iii) return it to the pickup location p_i. When an agent does not carry an obstacle (i.e., during step (i) or when it is not assigned with a task), we say that it is a *free* agent. When an agent does carry an obstacle (i.e., during steps (ii) and (iii)), we say that it is a *task* agent. Let $\tau = \langle p, d \rangle$ be a task assigned to agent a. We define the *pickup time* of τ as the first time step that a arrives at p. Similarly, we define the *drop-off time* of τ as the first time step that a returns to p after delivering the obstacle to d (namely, after the obstacle has been restored to its original location). The *service time* of τ in our setting is then defined to be the difference between its drop-off time and its pickup time.

As we can see, an assigned task τ has several states: enroute to pickup, delivery and restore. Therefore, it will be convenient to define a *goal mapping* λ to keep track of each agent's current goal. In other words, $\lambda(a, t, \tau)$ points to the next goal of agent a (being p or d) at timestep t.

Solution quality. A common measure of solution quality for a MAPD problem is *throughput* (Stern et al. 2019), defined as the average number of tasks completed per unit of time, which measures the amortized cost of completing all tasks. However, it is often important to consider costs that relate to individual tasks. Thus, we define the *ideal service time* to be the length of the shortest path between a task's pickup and delivery location (and back), while avoiding static obstacles and assuming there are no other agents. The task's *service time ratio* is then defined as the ratio between the task's actual *service time* and the task's *ideal service time*. Note that the closer the average service time ratio is to one, the closer the system is to its optimal throughput.

3.3 Terraforming MAPD with Disruptions

A tMAPD problem is defined identically to a MAPD problem with the difference that we allow free agents to move obstacles. That is, agents that are not currently carrying obstacles, can pick-up and drop-off obstacles with the purpose of opening passageways and reducing congestion. Additionally, we consider a setting where *disruptions* exist in the environment. Here, we define a disruption $\mathcal{D} = \langle v, t_{\text{start}}, t_{\text{end}} \rangle$ as the blockage of a vertex $v \in \mathcal{V}$ between timestep t_{start} and t_{end}. I.e., given such a disruption, no agent can pass through v between t_{start} and t_{end}. We assume that disruptions are unpredictable: at every timestep t the planner only has access to the currently active disruptions through a function OBSERVE. The function specifies all vertices $\mathcal{V}_{\mathcal{D}}$ that are blocked at t (i.e., vertices of disruptions where $t \in [t_{\text{start}}, t_{\text{end}}]$). This implies that the MAPD planner does not have access to (i) future disruptions or (ii) the time t_{end} for which an active disruption will end. In our setting, disruptions occur only along paths traversed by agents thus modelling realistic disruptions in warehouses such as items dropped from a pod or when agents malfunction.[2]

4 Algorithmic Background

In preparation to our approach for tMAPD, we first describe the PBS algorithm as a solver for the (classical) MAPF problem. We then describe how PBS can be used to solve the (standard) MAPD problem, as is often done in practice.

4.1 Priority-Based Search

We provide an overview of PBS together with an adaptation termed windowed PBS (W-PBS) for MAPF and refer the reader to Ma et al. (2019a) for further details. Pseudocode of W-PBS is detailed in Alg. 1.[3] At its core PBS takes a hierarchical approach using a high- and low-level search in which PBS maintains priorities between agents at the high-level and searches for agent paths that abide to these priorities in the low-level.

More specifically, in the high-level search, PBS explores a *priority tree* (PT), where a given node N of the PT encodes a (partial) priority set $P_N = \{a_h \prec a_i,\ a_j \prec a_l, \ldots\}$. A priority $a_i \prec a_j$ means that agent a_i has precedence over agent a_j whenever a low-level search is invoked (see below). In this case, we say that a_i has a higher priority than a_j. In addition to the ordering, each PT node maintains single-agent paths that represent the current MAPF solution (possibly containing collisions. The PBS algorithm (Alg. 1) starts the high-level search with the tree root whose priority set is empty, and assigns to each agent its shortest path (Lines 2-6). Whenever PBS expands a node N (Line 8), it invokes a low-level search to compute a new set of paths which abide by the priority set P_N. If a collision between agents, e.g., a_i and a_j, is encountered in the new paths, PBS generates two child PT nodes N_1, N_2 with the updated priority sets

[2] Our disruption model shares similarities to existing models in other domains such as railway planning (Mohanty et al. 2020).

[3] Blue text in Alg. 1 will be used to explain the adaptation of W-PBS to terraforming in Sec. 5 and should be ignored for now.

Algorithm 1: TW-PBS

Input: Graph $\mathcal{G}$, agents $\mathcal{A}$, **movable obstacles** $\tilde{\mathcal{O}}$, goal mapping λ, planning window ω
Returns: A collision-free plan $\boldsymbol{\pi}$

1 $\mathcal{A} \leftarrow \mathcal{A} \cup \tilde{\mathcal{O}}$ // treat $\tilde{\mathcal{O}}$ as demi-agents
2 $R.priorities \leftarrow \emptyset$ // root state
3 $R.paths \leftarrow \text{findPaths}(\mathcal{G},\ \mathcal{A},\ \lambda,\ R.priorities, \omega)$
4 $R.cost \leftarrow \text{get}\mathbf{Terra}\text{Flowtime}(R.paths)$
5 $R.collisions \leftarrow \text{detectCollisions}(R.paths)$
6 $\text{insert}(R, \text{OPEN})$
7 **while** OPEN *not empty* **do**
8 $N \leftarrow \text{pop}(\text{OPEN})$ // searched via DFS
9 $\langle a_i, a_j, l, t\rangle \leftarrow \text{getCollisions}(N)$
10 **if** $N.collisions$ *is empty* **then**
11 **return** $N.paths$
12 **for** $p \in \langle a_i \prec a_j\rangle, \langle a_j \prec a_i\rangle$ **do**
13 $P \leftarrow N.priorities \cup \{p\}$
14 $N' \leftarrow \text{clone}(N)$
15 $N'.paths \leftarrow \text{findPaths}(\mathcal{G},\ \mathcal{A},\ \lambda,\ P,\ \omega)$
16 $N'.cost \leftarrow \text{get}\mathbf{Terra}\text{Flowtime}(N'.paths)$
17 $N'.collisions \leftarrow$ $\text{detectCollisions}(N'.paths)$
18 $N'.priorities \leftarrow P$
19 $\text{insert}(N', \text{OPEN})$

$P_{N_1} = P_N \cup \{a_i \prec a_j\}, P_{N_2} = P_N \cup \{a_j \prec a_i\}$, respectively (Line 12). The high-level search chooses to expand at each iteration a PT node in a depth-first search manner. The high-level search terminates when a valid solution is found at some node N (Line 10), or when no more nodes for expansion remain, in which case, PBS declares failure.

The low-level search of PBS proceeds in the following manner. For a given PT node N, PBS performs a topological sort of the agents according to P_N from high priority to low, and plans individual-agent paths based on the ordering. For a given topological ordering $(a'_1, \ldots, a'_{k'}) \subset \mathcal{A}$, for some $1 \leq k' \leq k$, the low-level iterates over the k' agents in the topological ordering, and updates their paths such that they do not collide with any higher-priority agents. Note that agents that do not appear on this list maintain their original plans. It then checks all agents for any remaining collisions.

As we will see shortly, to speed up planning times, it is often convenient to consider collisions only for the first ω timesteps. As previously mentioned, this variant is called *windowed*-PBS or W-PBS for short.

4.2 Solving MAPD using RHCR

The common approach to solve MAPD problems (Li et al. 2021b; Okumura, Tamura, and Défago 2021) is by iteratively (i) assigning tasks to free agents, (ii) deriving target locations from these tasks and setting starting locations to be the agents' current locations and (iii) running a MAPF solver to compute collision-free paths for all the agents. The first step, task assignment, can be solved in a variety of methods (see, e.g., (Ma et al. 2017, 2019b)) but in this work we limit ourselves to greedy task assignment. Specifically, we compute the graph distance (i.e., the number of edges in a shortest path) of each agent to each goal and assign tasks greedily according to these distances. The last step, solving a MAPF problem, is typically done by computing collision-free paths up to a certain time-horizon hence running a windowed MAPF solver such as W-PBS. These aspects form the RHCR algorithm (Li et al. 2021b) which demonstrates state-of-the-art performance by re-planning agent paths in regular periods of h simulation timesteps and resolving inter-agent collisions occurring within a time window w, such that $w \geq h$. The use of bounded-horizon PBS (i.e., W-PBS) yields high throughput at a reduced computation effort, albeit without completeness or optimality guarantees.

A detail that is crucial to iteratively applying W-PBS as tasks arrive in an online manner, is the careful handling of priority ordering. Namely, whenever an agent is assigned a new task, or completes a sub-task, its path is replanned with its priority ordering wiped, to ensure it will not give automatic precedence over other agents based on past interactions. After each time step, the planning horizon is extended and paths are replanned until all tasks are complete.

5 Algorithmic Framework

In this section we present our algorithmic framework for incorporating terraforming into MAPD. Our approach follows the RHCR approach (Sec. 4.2) to solve MAPD, i.e., we assign new tasks to agents and then decompose the problem into a sequence of MAPF instances to use W-PBS to solve these individual queries. Subsequently, we observe the environment to detect if there are disruptions. When disruptions are observed, we start by planning new paths for agents (without terraforming). We then consider if it is worthwhile to invoke terraforming to manipulate the environment. To do so, we initiate a simplified terraforming MAPF (tMAPF) problem (defined below) where the set of movable obstacles is in the local neighborhood of the disruption, and the movable obstacles are *self-propelled*, i.e., can move without the intervention of agents. This simplification allows us to defer the question of who manipulates an obstacle to a later stage, and momentarily only reason about whether obstacles should be moved at all. The solution to this tMAPF problem does not fully account for the cost of moving the obstacles and is used to evaluate which obstacles (if any) should be displaced. Subsequently, new tasks are defined and assigned to agents corresponding to obstacles that should be displaced as specified by the tMAPF solution.

For simplicity, we assume that terraforming tasks are structured similarly to standard tasks (i.e., there is a delivery location to which the obstacle will temporarily be moved to). Thus, in addition to the pickup location of each task, which corresponds to the location of an obstacle, a delivery location must be determined. To this end, we assume there is a defined subset of vertices within the graph $\hat{\mathcal{V}} \subset \mathcal{V}$, exclusively reserved for this purpose. When a new terraforming task is created, the nearest vertex from this subset is selected as the task's delivery location.

The rest of the section formalizes and details our approach. Specifically, we start with detailing the high-level

Algorithm 2: getTerraformingTasks

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, affected agents $\tilde{\mathcal{A}}$, self-propelled obstacles $\tilde{\mathcal{O}}$, goal mapping λ, planning window ω, terraforming reserved locations $\hat{\mathcal{V}}$

Returns: A set of Terraforming tasks $\tilde{\mathcal{T}}$

1 $\pi_{\text{MOVABLE}} \leftarrow$ TW-PBS$(\mathcal{G}, \tilde{\mathcal{A}}, \tilde{\mathcal{O}}, \lambda, \omega)$
2 $\tilde{\mathcal{O}}_{\text{MOVED}} \leftarrow \text{getObstaclesToMove}(\pi_{\text{MOVABLE}})$
3 $\tilde{\mathcal{T}} \leftarrow \text{convertToTasks}(\tilde{\mathcal{O}}_{\text{MOVED}}, \hat{\mathcal{V}})$
4 return $\tilde{\mathcal{T}}$

approach for tMAPD, which we call tRHCR, and then continue to describe the individual components such as an adaptation of W-PBS to terraforming.

5.1 Terraforming RHCR

We are ready to detail our algorithmic framework tRHCR for tMAPD (Alg. 3). Recall that at every timestep and while tasks remain (Line 2), we assign tasks to agents (Line 3) and replan paths for agents that obtain new tasks or agents whose path requires re-planning to ensure a sufficient planning horizon (Line 4). We then observe for disruptions (Line 5), which is where our approach deviates from RHCR.[4]

As mentioned, disruption detection is done using the OBSERVE function, which returns a set $\mathcal{V}_{\mathcal{D}} \subset \mathcal{V}$ of all newly detected disruption locations at the current timestep t (recall that we do not have access to the termination time of the disruptions). When disruptions occur (Lines 6-15), we start by computing the set of agents $\tilde{\mathcal{A}}$ that are *affected* by the disruptions using the routine getAffectedAgents (see details below). Informally, an agent a_i is considered to be affected if its path gets blocked due to a disruption. Additionally, we include any agent a_j that gives a_i precedence (i.e., $a_i \prec a_j$) as a_j may have altered its desired path due to a restriction imposed by a_i's path. We then call W-PBS (Line 8) to replan paths for the affected agents while preserving the paths of those unaffected. Note that Terraforming does not take place at this stage.

To consider Terraforming, we start (Line 9) by computing a set of candidate obstacles $\tilde{\mathcal{O}}$ that are evaluated for displacement. The set $\tilde{\mathcal{O}}$ is defined to be all obstacles within a graph distance of $r \geq 1$ from a disruption, where r is called the *terraforming radius*.[5] Namely,

$$\tilde{\mathcal{O}} := \{o \in \mathcal{O} \mid \exists v \in \mathcal{V}_{\mathcal{D}} \text{ s.t. } \|o, v\| \leq r\}.$$

In the next step, we identify which obstacles from $\tilde{\mathcal{O}}$ could be potentially displaced in order to clear the way for the affected agents $\tilde{\mathcal{A}}$ and where those should be displaced to.

[4]Note that if path planning took place after observing new disruptions, all agents would avoid these disruptions. By planning paths before checking for disruptions, agents are unaware of new obstructions and assume their desired path will be available allowing us to identify which agents are affected by new disruptions.

[5]In the case of a grid, the graph distance (the number of edges in a shortest path connecting two vertices) is the Manhattan distance.

Algorithm 3: tRHCR

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, agents $\mathcal{A}$, obstacles $\mathcal{O}$, tasks stream $\mathcal{T}$, terraforming radius r, planning window ω, terraforming reserved locations $\hat{\mathcal{V}}$

Returns: A collision-free MAPD plan $\boldsymbol{\pi}_{\text{MAPD}}$

1 $t \leftarrow 0$
2 **while** $\mathcal{T}$ *is not empty* ***or*** *agents have tasks* **do**
3 $\quad \lambda \leftarrow \text{assignTasks}(\mathcal{A}, \mathcal{T})$
4 $\quad \boldsymbol{\pi} \leftarrow$ W-PBS$(\mathcal{G}, \mathcal{A}, \lambda, \omega)$
5 $\quad \mathcal{V}_{\mathcal{D}} \leftarrow \text{OBSERVE}(\mathcal{G}, t)$
6 $\quad$ **if** $\mathcal{V}_{\mathcal{D}}$ *is not empty* **then**
7 $\quad\quad \tilde{\mathcal{A}} \leftarrow \text{getAffectedAgents}(\boldsymbol{\pi}, \mathcal{V}_{\mathcal{D}})$
8 $\quad\quad \boldsymbol{\pi} \leftarrow$ W-PBS$(\mathcal{G} \setminus \mathcal{V}_{\mathcal{D}}, \tilde{\mathcal{A}}, \lambda, \omega)$
9 $\quad\quad \tilde{\mathcal{O}} \leftarrow \text{candidateObstacles}(\mathcal{O}, \mathcal{V}_{\mathcal{D}}, r)$
10 $\quad\quad \tilde{\mathcal{T}} \leftarrow \text{getTerraformingTasks}(\mathcal{G}, \tilde{\mathcal{A}}, \tilde{\mathcal{O}}, \lambda, \omega, \hat{\mathcal{V}})$
11 $\quad\quad \tilde{\lambda} \leftarrow \text{assignTasks}(\mathcal{A}, \tilde{\mathcal{T}} \cdot \mathcal{T})$
12 $\quad\quad \boldsymbol{\pi}_{\text{TERRA}} \leftarrow$ W-PBS$(\mathcal{G}, \mathcal{A}, \tilde{\lambda}, \omega)$
13 $\quad\quad$ **if** $|\boldsymbol{\pi}_{\text{TERRA}}| < |\boldsymbol{\pi}|$ **then**
14 $\quad\quad\quad \boldsymbol{\pi} \leftarrow \boldsymbol{\pi}_{\text{TERRA}}$
15 $\quad\quad\quad \mathcal{T} \leftarrow \tilde{\mathcal{T}} \cdot \mathcal{T}$
16 $\quad \boldsymbol{\pi}_{\text{MAPD}}$.append actions from $\boldsymbol{\pi}$ at timestep t
17 $\quad t \leftarrow t + 1$

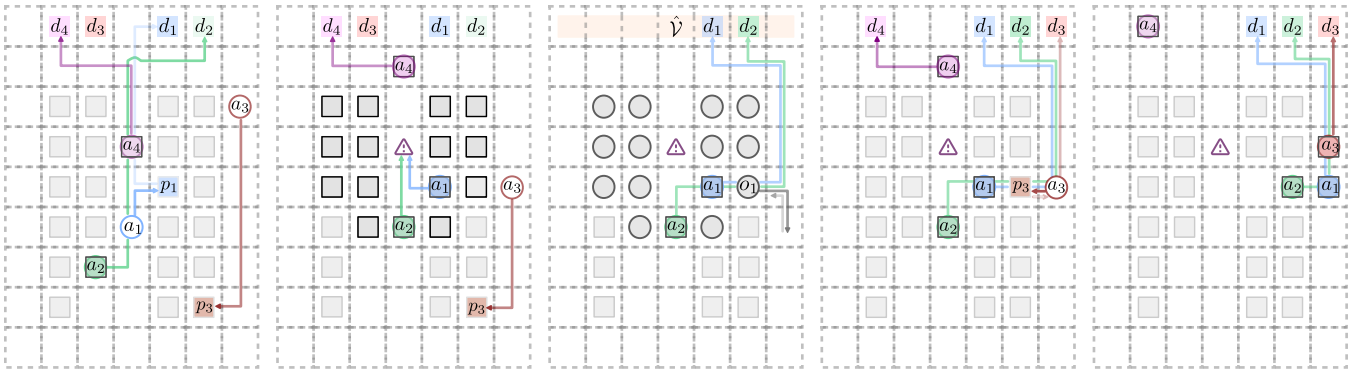
This is done by calling the getTerraforimingTasks subroutine in Line 10 (detailed in Sec. 5.2), which returns a new set of tasks $\tilde{\mathcal{T}}$ that specify which obstacles from $\tilde{\mathcal{O}}$ should be moved, and where to move it.

In the next step (Line 11), we compute a new assignment $\tilde{\lambda}$ for the agents $\mathcal{A}$. Here we assign both the new tasks $\tilde{\mathcal{T}}$ which may require agents to move obstacles in $\tilde{\mathcal{O}}$ and the original tasks $\mathcal{T}$ (we use $\tilde{\mathcal{T}} \cdot \mathcal{T}$ to denote the concatenation of the two task sets). Note that the new tasks $\tilde{\mathcal{T}}$ are assigned before the existing tasks $\mathcal{T}$ and that only the first $|\mathcal{A}|$ tasks are assigned. After the new assignment was defined, a new MAPF problem is defined and solved (Line 12) to obtain a solution $\boldsymbol{\pi}_{\text{TERRA}}$. Finally, we choose $\boldsymbol{\pi}$ to be the lowest-cost solution among the options of not using and using terraforming (Lines 13-14).

5.2 Details for getTerraformingTasks

We provide details on the getTerraformingTasks subroutine. First, we describe the tMAPF instance it solves within, our solution approach for tMAPF, which we call TW-PBS, and the steps in Alg. 2.

tMAPF. We define the tMAPF problem as a generalization to the MAPF problem, where in addition to the graph $\mathcal{G}$, agents $\mathcal{A}$ and goal mapping λ, the input includes a subset of obstacles $\tilde{\mathcal{O}} \subset \mathcal{O}$ as movable obstacles. An obstacle $o \in \tilde{O}$ needs to be restored to its original position to avoid a permanent alteration of the environment. Recall that we make the simplifying assumption that the movable obstacles $\tilde{\mathcal{O}} \subset \mathcal{O}$



(a) t (b) $t+2$ (c) tMAPF instance (d) $\boldsymbol{\pi}_{\text{TERRA}}$ (e) $t+5$

Figure 2: tRHCR visualization (see description in Sec. 5.3).

are *self-propelled*, i.e., they can move on their own if necessary. Hence we will refer to them as "demi-agents".

Intuitively, demi-agents are considered as already waiting at their goal location and regular agents may collide with them. This will cause either the colliding agent to recompute its path or cause the demi-agent to move in order to allow the agent to pass. The difference between costs with and without terraforming lies in how we compute the cost of a demi-agent's path. In contrast to a (regular) agent's path π whose cost $|\pi|$ is the number of timesteps taken to execute π, for a demi-agent's path $\tilde{\pi}$, we define its cost $|\tilde{\pi}|$ as the number of movements taken by the demi-agent (i.e., wait actions do not incur a cost, so that idle obstacles do not incur a penalty cost). Without this modification, performing terraforming would not be worthwhile as all movable obstacles would incur cost, regardless of whether they moved or not. To this end, with a slight abuse of notation, we denote the cost of a solution as $|\boldsymbol{\pi}| = \sum_{i \in \mathcal{A}} |\pi_i| + \sum_{j \in \tilde{\mathcal{O}}} |\tilde{\pi}_j|$ and refer to it as the *terra-flowtime*.

Solving tMAPF with TW-PBS. Alg. 1 outlines the pseudocode of TW-PBS with the differences from W-PBS highlighted in **bold**. Specifically, the set of movable obstacles $\tilde{\mathcal{O}}$ are treated as "demi-agents" and added to the set of agents $\mathcal{A}$ to create the set of agents the algorithm considers (Line 1). When the cost of a node is computed (Line 4 and 16), instead of computing the (standard) flowtime via the function getFlowtime, we compute the terraforming *flowtime* via the function getTerraFlowtime that does not account for movable obstacles' wait actions.

Wrapping up. We finalize the details of Alg. 2. We start (Line 1) by solving the tMAPF problem that allows obstacles to move upon collision with agents and let π_{MOVABLE} be the solution to this problem. We then extract the set of obstacles $\tilde{\mathcal{O}}_{\text{MOVED}}$ that are required to move as part of π_{MOVABLE} (Line 2) and use them to define a new set of tasks $\tilde{\mathcal{T}}$ (Line 3), defined as follows. For each obstacle that needs to be moved $o \in \tilde{\mathcal{O}}_{\text{MOVED}}$, we generate a new task $\langle o, \ell \rangle$, where $o \in \mathcal{O}$ (the obstacle's location) is the task's pickup location and $\ell \in \hat{\mathcal{V}}$ is the delivery location. ℓ is chosen to be the point nearest to the pickup location o in the reserved location set. Upon being assigned the task, an agent will move the obstacle to perform terraforming, before returning it to its original location. The newly generated set of tasks $\tilde{\mathcal{T}}$ is then returned as output (Line 4). Note that the computed path of a self-propelled obstacle in the tMAPF problem is not used, but rather it only serves as a hint on whether it pays off to move the obstacle in the first place.

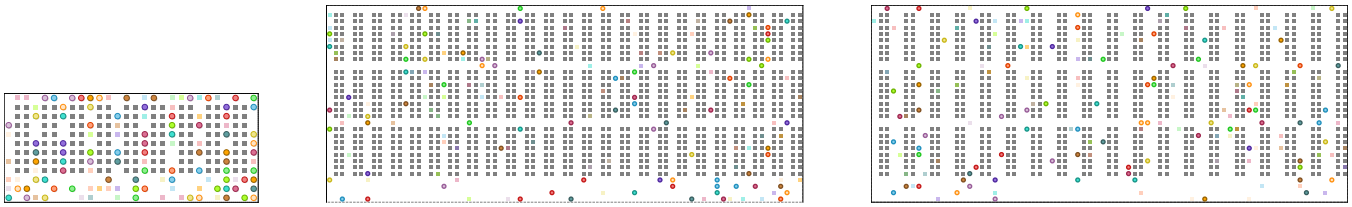
5.3 Additional Details

We give additional details on algorithmic building blocks described in Sec. 5.1 and an example of the algorithm's flow.

Affected agents. The function getAffectedAgents (invoked in Alg. 3, Line 7 computes the set of agents $\tilde{\mathcal{A}}$ that are affected by the disruptions. An agent is considered to be affected by a disruption if (i) their path is blocked by a disruption or if (ii) an agent with higher priority is affected by the disruption. The second condition is important to allow agents whose path was constrained by higher-priority agents to re-plan their path after the higher-priority agents replanned their own paths.

This set can be computed straightforwardly by adding all agents $a_i \in \mathcal{A}$ whose path $\pi_i \in \boldsymbol{\pi}$ intersects a vertex in $\mathcal{V}_{\mathcal{D}}$. All these agents are pushed into a stack S and as long as S is not empty, the algorithm pops an agent a, collects all agents with higher priority and if they are not in $\tilde{\mathcal{A}}$, adds them to $\tilde{\mathcal{A}}$ and to the stack S.

Example. We provide in Fig. 2 a visualization of tRHCR. **Fig. 2a**: at time t we have two task agents (a_2 and a_4) and two free agents (a_1 and a_3). Agents a_1 and a_3 are assigned new tasks (Alg. 3, Line 3) and W-PBS is used to plan suitable paths (Alg. 3, Line 4). For each agent, we trace its path to its next goal (pick up p_i or delivery d_i). **Fig. 2b**: at time $t+2$ (i.e., after two timesteps), a disruption is detected (Alg. 3, Line 5) due to an item dropped by agent a_4 along its path. The affected agents (Line 7) are a_1 and a_2, as the disruption blocks their paths. We then use W-PBS to replan paths for the affected agents (Alg. 3, Line 8) which would produce lengthy detours around the disruption. Subsequently, a set of candidate obstacles to be moved (Line 9) is computed for $r = 3$, illustrated with bold boundary. Then follows a call to getTerraformingTasks (Line 10). **Fig. 2c**: to compute the terraforming tasks (Alg. 2), we form



(a) (b) (c)

Figure 3: Warehouses used in our empirical evaluation. (a), (b) and (c) depict the MEDIUM, LARGE and LARGE2WIDE environments, with 80 task agents (colorful dots). Here, rows of pods (gray rectangles) form long narrow aisles and goals (colorful rectangles) are selected from nearby workstations around the borders of each map.

a tMAPF instance that includes the affected agents and the (self-propelled) candidate obstacles (Line 1). Paths computed are traced with a line. Note that one self-propelled obstacle o_1 moved which is returned by getObstaclesToMove (Line 2). Next, we create a terraforming task (Line 3), where the pickup location is the location of obstacle o_1 and the delivery location is the closest location that is reserved for terraforming: the reserved locations $\hat{\mathcal{V}}$, are highlighted in orange at the top of the map. **Fig. 2d**: Returning to Alg. 3, Line 10 with the new terraforming task (Alg. 2), we assign this new task to agent a_3 (Line 11) hence postponing its current task. The new plan $\boldsymbol{\pi}_{\text{TERRA}}$ (Line 12) is outlined with the extraction of the obstacle and agents making use of the opening. **Fig. 2e**: At time $t+5$ (after three additional timesteps), agents a_1, a_2 and a_3 continue towards their delivery location. Agent a_3 carries the obstacle away to maintain existing pathways clear and keep the shortcut open. Agent a_4 reaches its respective task-delivery location and its next goal will be returning the obstacle's pickup location.

6 Evaluation

We evaluate our approach using maps inspired by autonomous warehouses: a MEDIUM 24×47 map (Felner et al. 2018; Li et al. 2019), a LARGE 32×75 map (Li et al. 2020) and a LARGE2WIDE 32×75 map similar to LARGE but with corridors that are 2-obstacle wide (see Fig. 3).

Unless stated otherwise, for each map we (i) run 25 random instances each with 600 randomly generated pickup and delivery tasks, (ii) use terraforming radius of $r=8$ and 100 agents, (iii) consider two causes of disruptions (an immobilized agent and a dropped item) where each type may occur along an agent path with probability of 0.5% per simulation step and block a location for a random time interval in the range of $[40,60]$ simulation steps.[6] Finally, our algorithms are implemented in Python and tested on an Ubuntu machine with 16GB RAM and a 2.7GHz Intel i7 CPU.

Throughput. We report a comparison of the throughput, which is the average number of tasks completed per simulation step, for RHCR and tRHCR. We present (Fig. 4a) the average throughput on each map for a varying number of agents. Note that for all maps and values of $|\mathcal{A}|$, terraforming increases the average throughput by between 7% and 11%.

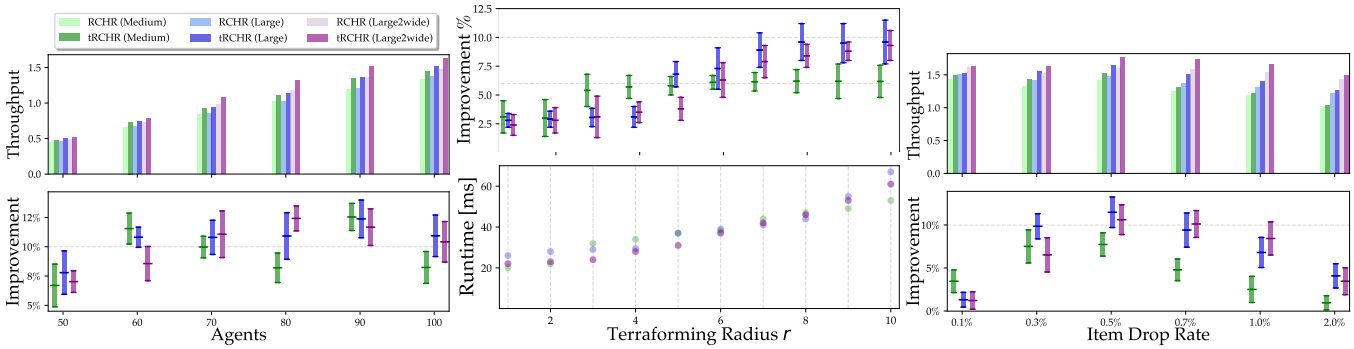
[6]These numbers were verified by Amazon representative in private communication as being realistic values.

Runtime with and without disruptions. The lion's share of our RHCR algorithms (with and without terraforming) is the MAPF planner which in our case is either W-PBS or TW-PBS. To this end, we report (Fig. 5, top) the running time of each iteration of RHCR and tRHCR, for a sample scenario with 100 agents in the LARGE environment.

When no disruptions occur, the two algorithms are essentially identical and indeed their running time is roughly the same (it is not identical as terraforming causes agents' path to differ affecting future timesteps of the simulation). When disruptions occur (vertical dashed lines in Fig. 5, top), then planning times for TW-PBS increases by an order of magnitude. This is not surprising as the number of agents (including candidate obstacles as "demi-agents") that TW-PBS considers (which is the number of affected agents and candidate obstacles) is substantially larger than the number of agents that W-PBS considers (which is only the number of affected agents). This can be seen in the bottom of Fig. 5 in which TW-PBS is applied on up to $10\times$ more agents than W-PBS. Having said that, when amortizing the running time of TW-PBS over the entire simulation, the average computation time, is still within the arguably tolerable range of $40ms$ per planning window, compared with the average runtime of W-PBS of $5ms$. Moreover, the improvement in throughput as previously discussed and demonstrated in Fig. 4a can make the extra runtime worthwhile.

Maximum task service time ratio. Recall that the service time is the number of timesteps elapsed between a task's pick-up time and its completion (drop-off) time, whereas the ideal service time is the shortest service time attainable for a task when no other agents (or disruptions) interfere with its execution. For each task, the ratio between the former and the latter is a measure of task delay. We compare (Fig. 5, middle) the maximal service time ratio for both RHCR and tRHCR. Note that Fig. 5 shows the service time of the most-delayed task, while the average tasks' delay is reflected in the throughout as shown in Fig. 4a. This allows to pinpoint where the increase in throughput (Fig. 4a) comes from and to justify the need to perform the expensive operation of terraforming when disruptions occur (Fig. 5, top).

Indeed, we can see that the maximal service time ratio for RHCR is roughly $2\times$ to $3\times$ larger than the maximal service time ratio of tRHCR. This is expected as agents that are blocked by disruptions require detouring obstructed vertices and in doing so increase their service time. To a greater extent, agents that become trapped by surrounding disruptions



(a) (b) (c)

Figure 4: (a) Throughput of MAPD scenarios with disruptions, comparing the RHCR-based baseline approach to tRHCR. Top and bottom plots show the absolute throughput and improvement (i.e., ratio of tRHCR's throughput with the RHCR-based throughput), respectively. Here, error bars denote one standard deviation. (b) Evaluation of Terraforming radius r on throughput improvement (top) and average runtime (bottom) per simulation timestep of tRHCR. Shaded margins in the top figure corresponding to one standard deviation. (c) Throughput as a function of disruption rate of dropped items.

can only wait in place until an opening is cleared. In contrast, terraforming creates pathways (when beneficial) and shortcuts that decrease the service time of blocked agents.

Terraforming radius evaluation. Recall that the terraforming radius r is used to compute the set of candidate movable obstacles $\tilde{\mathcal{O}}$ (Alg. 3, Line 9). The number of candidate movable obstacles $|\tilde{\mathcal{O}}|$, and thus the complexity of TW-PBS, grows proportionally with r. However, the potential improvement in solution quality grows with r as well since terraforming has more options to displace obstacles. Thus, we evaluate both the throughput improvement (Fig. 4b, top) and the algorithm's runtime (Fig. 4b, bottom) as a function of the terraforming radius. We can see the tradeoff between the increased computation time and the improved throughput that happens as r increases. Empirically $r = 8$ balances between the two in all environments.

Evaluating item drop rate. We evaluate the effect of disruption rates on throughput. Specifically, we consider varying rates of disruptions caused by a dropped item, and maintain agent breakdown rate of 0.5% per timestep (we fix disruption rates caused by agent breakdowns because higher disruption rates do not leave enough agents to perform terraforming and to complete tasks). Results, summarized in Fig. 4c show that as disruption rates caused by dropped items increase, so does the throughput improvement, which peaks at 0.5% disruptions per timestep. This improvement then decreases as (i) there may be more disruptions than can be handled by free agents or (ii) alternative paths offered by obstacle displacements become blocked as well.

7 Discussion and Future Work

In this work we explored the potential benefits of terraforming within MAPD, with emphasis on mitigating the impact of unforeseen disruptions. The extreme case of an agent becoming trapped by disruptions offers the most convincing motivation for terraforming. Without terraforming, such agents cannot make any progress while terraforming allows to maintain (and even improve) throughput by having nearby

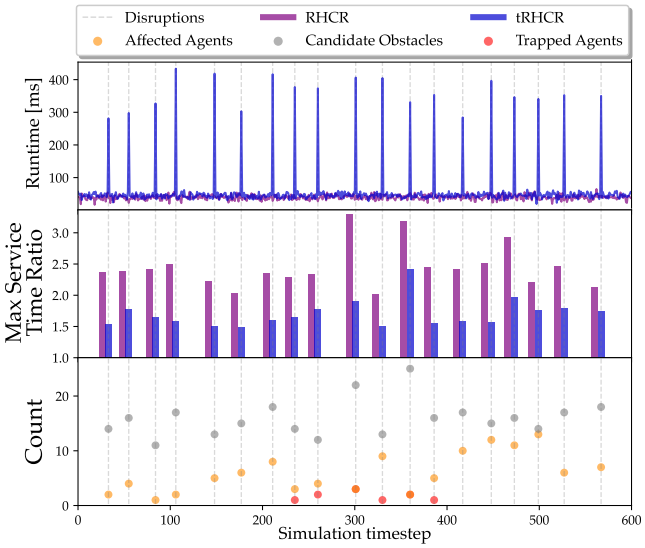


Figure 5: (Top) Runtime (in ms) of RHCR and tRHCR per simulation timestep with disruptions marked by dashed lines. (Middle) The maximal task service time ratio for both RHCR and tRHCR when disruptions occur (lower is better). (Bottom) The number of agents affected by disruptions (Alg. 3, Line 8), the number of candidate obstacles submitted to TW-PBS (Alg. 3, Line 9) and the number of trapped agents due to disruptions.

agents create shortcuts and reduce per-task execution time.

To harness the full potential of tMAPF, we envision its application to MAPD where agents en-route to collect an item can optimize the environment through local changes that serve multiple agents, regardless of disruptions. As future work we wish to consider an approach that facilitates subtle environment manipulation with little associated cost, but with enough foresight as to yield a significant benefit to nearby agents, which could provide a substantial boost to overall throughput. Furthermore, we plan to formally analyze the new variant's complexity in future work.

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2018. Robust multi-agent path finding. In *Symposium on Combinatorial Search (SoCS)*.

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2020. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research*, 67: 549–579.

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 *European Conference on Artificial Intelligence (ECAI)*, volume 263, 961–962.

Bellusci, M.; Basilico, N.; and Amigoni, F. 2020. Multi-Agent Path Finding in Configurable Environments. In *Autonomous Agents and MultiAgent Systems (AAMAS)*, 159–167.

Belov, G.; Du, W.; de la Banda, M. G.; Harabor, D.; Koenig, S.; and Wei, X. 2020. From Multi-Agent Pathfinding to 3D Pipe Routing. In Harabor, D.; and Vallati, M., eds., *Symposium on Combinatorial Search (SoCS)*, 11–19.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, S. E. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 740–746.

Choudhury, S.; Solovey, K.; Kochenderfer, M. J.; and Pavone, M. 2021. Efficient Large-Scale Multi-Drone Delivery using Transit Networks. *Journal of Artificial Intelligence Research*, 70: 757–788.

Erdem, E.; Kisa, D. G.; Oztok, U.; and Schüller, P. 2013. A general formal framework for pathfinding problems with multiple agents. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. S.; and Koenig, S. 2018. Adding heuristics to conflict-based search for multi-agent path finding. *International Conference on Automated Planning and Scheduling (ICAPS)*, 28: 83–87.

Greshler, N.; Gordon, O.; Salzman, O.; and Shimkin, N. 2021. Cooperative Multi-Agent Path Finding: Beyond Path Planning and Avoidance. In *Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, 20–28.

Li, J.; Chen, Z.; Zheng, Y.; Chan, S.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2021a. Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 477–485.

Li, J.; Gange, G.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2020. New techniques for pairwise symmetry breaking in multi-agent path finding. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, 193–201.

Li, J.; Harabor, D.; Stuckey, P. J.; Felner, A.; Ma, H.; and Koenig, S. 2019. Disjoint splitting for multi-agent path finding with conflict-based search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 29, 279–283.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J.; Kumar, S.; and Koenig, S. 2021b. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Liu, M.; Ma, H.; Li, J.; and Koenig, S. 2019. Task and path planning for multi-agent pickup and delivery. *Autonomous Agents and MultiAgent Systems (AAMAS)*, 12: 206–208.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019a. Searching with consistent prioritization for multi-agent path finding. In *Association for the Advancement of Artificial Intelligence (AAAI)*, volume 33, 7643–7650.

Ma, H.; Hönig, W.; Kumar, T. S.; Ayanian, N.; and Koenig, S. 2019b. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Association for the Advancement of Artificial Intelligence (AAAI)*, volume 33, 7651–7658.

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

Madar, N.; Solovey, K.; and Salzman, O. 2022. Leveraging Experience in Lifelong Multi-Agent Pathfinding. In *Symposium on Combinatorial Search (SoCS)*, 118–126.

Mohanty, S.; Nygren, E.; Laurent, F.; Schneider, M.; Scheller, C.; Bhattacharya, N.; Watson, J.; Egli, A.; Eichenberger, C.; Baumberger, C.; et al. 2020. Flatland-rl: Multi-agent reinforcement learning on trains. *arXiv preprint arXiv:2012.05893*.

Okumura, K.; Tamura, Y.; and Défago, X. 2021. Iterative Refinement for Real-Time Multi-Robot Path Planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9690–9697.

Salzman, O.; and Stern, R. 2020. Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems. In *Autonomous Agents and MultiAgent Systems (AAMAS)*, 1711–1715.

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

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, S.; et al. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search (SoCS)*, 10: 151–158.

Surynek, P.; Felner, A.; Stern, R.; and Boyarski, E. 2016. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *European Conference on Artificial Intelligence (ECAI)*, 810–818. IOS Press.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29(1): 9.

Yu, J.; and LaValle, S. M. 2012. Multi-agent Path Planning and Network Flow. In *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, volume 86, 157–173. Springer.