Optimal Multi-Agent Pickup and Delivery Using Branch-and-Cut-and-Price

Edward Lam, Peter J. Stuckey, and Daniel Harabor

Monash University

July 31, 2023

Abstract

Given a set of agents and a set of pickup-delivery requests located on a two-dimensional map, the Multi-Agent Pickup and Delivery problem assigns the requests to the agents such that every agent moves from its start location to the locations of its assigned requests and finally to its end location without colliding into any other agent and that the sum of arrival times is minimized. This paper proposes two exact branch-and-cut-and-price algorithms for the problem. The first algorithm performs a three-level search. A high-level master problem selects an optimal sequence of requests and a path for every agent from a large collection. A mid-level sequencing problem and a low-level navigation problem are simultaneously solved to incrementally enlarge the collection of request sequences and paths. The second algorithm first solves the sequencing problem to find a set of request sequences and then solves the navigation problem to determine if paths compatible with the request sequences exists. Experimental results indicate that the integrated algorithm solves more instances with higher congestion, and the deferred algorithm solves more instances with lower congestion and could scale to 100 agents and 100 requests in one case.

1 Introduction

The Multi-Agent Pickup and Delivery (MAPD) problem is an abstraction of the problem of controlling robots in automated warehouses. The problem is defined on a discrete time horizon and a two-dimensional map discretized into square cells called locations. A set of cooperating agents is situated on the map, each associated with a fixed start location and end location. At every timestep, an agent can move north, south, east or west, or wait at its current location. Some locations are designated as obstacles, which agents cannot pass through.

The problem considers a set of orders. Each order consists of a pickup request and a delivery request. Every pickup request and delivery request is associated with a location and a time window. Every order must be assigned to an agent. Every agent must depart its start location, visit the locations of the pickup and delivery requests of its assigned orders within their time windows and then arrive at its end location before the end of the planning period. Once an agent starts a pickup, the agent must then complete the associated delivery before starting another pickup. This limitation models robots that can only carry one item at any given time.

The agents must not collide into each other while traveling. At most one agent can be at a location at any given time, called the vertex collision condition. Agents also cannot cross over each other into opposite locations, called the edge collision condition. The time that an agent reaches its end location after completing all its assigned orders, if any, and waits there indefinitely (because it no longer needs to move out of the way for other agents to pass through its end location) is called its end time. The problem attempts to assign the orders to the agents and find paths for the agents to visit the locations of their assigned pickup and delivery requests that minimize the sum of end times, i.e., the so-called sum-of-costs objective.

This paper presents two optimal algorithms for MAPD, named BCP-MAPD and BCPB-MAPD. Both algorithms are based on branch-and-cut-and-price, a mathematical programming technique that dynamically builds the variables and constraints of a linear relaxation for computing a lower bound within a branch-and-bound tree search.

BCP-MAPD can be conceptually viewed as a three-level search. A high-level master problem selects a sequence of requests and a path on the map for every agent from a large set, while ensuring that the agents do not collide. A pricing problem incrementally builds the set of request sequences and paths in the master problem by simultaneously solving a mid-level sequencing problem and a low-level navigation problem. The sequencing problem determines a sequence of requests for an agent and the navigation problem determines a path on the map directing the agent to the location of each successive request in the sequence.

BCPB-MAPD relies on similar ideas but first optimizes the sequences before optimizing the paths, instead of simultaneously optimizing both the sequences and the paths. The master problem selects a sequence of requests for every agent from a large set, which is dynamically constructed by a pricing problem. Whenever a feasible set of sequences is found, a discrete Benders problem checks if these sequences yield feasible paths on the map. If the sequences are infeasible or superoptimal with respect to the path finding, a combinatorial feasibility cut or optimality cut is added to the master problem, forcing it to choose a different set of sequences.

Experimental results indicate that neither algorithm dominates. The joint optimization algorithm achieves an average optimality gap of 0.2%, solves more instances with higher congestion and could scale to 20 agents and 50 orders on a warehouse map. The deferred path finding algorithm achieves an average optimality gap of 4.0%, solves more instances with lower congestion and could reach 100 agents and 100 orders on a computer game map.

The contributions of this paper are (1) two optimal algorithms for MAPD, (2) a pricing algorithm for solving the two-level simultaneous sequencing and path finding problem and (3) computational results showing that exact mathematical techniques are competitive and even surpass methods from the artificial intelligence community, where the MAPD problem first appeared.

2 Background and Literature Review

Liu et al. (2019) recognized that MAPD combines elements of the Multi-Agent Path Finding (MAPF) problem (e.g., Stern et al. 2019) and the Pickup and Delivery Problem with Time Windows (PDPTW) from the family of Vehicle Routing Problems (VRPs) (e.g., Vigo and Toth 2014).

The PDPTW is a sequencing problem of assigning pickup-delivery orders to agents, called vehicles. Each order is broken up into a pickup request and a delivery request. The PDPTW assigns the orders to the vehicles such that every request is completed within its time window and the total travel distance of all vehicles is minimized. All vehicles start at a central depot, visit the locations of their assigned requests and then return to the depot. Collisions are not considered in the PDPTW because the network is defined at a coarser level. All current state-of-the-art exact algorithms for the PDPTW are based on branch-and-cut-and-price. Dumas, Desrosiers, and Soumis (1991) developed the first branch-and-price algorithm for the PDPTW, which did not include valid inequalities, and used a branching rule that generated many children at each node. Røpke and Cordeau (2009) extended this algorithm with several families of cutting planes and proposed two pricing problems that generate different classes of paths. In the first variant, the elementarity constraint (each request can be completed at most once along a path) is enforced in both the master problem and the pricing problem. In the second variant, the elementarity constraint is only enforced in the master problem, giving rise to an easier pricing problem but a weaker dual bound. Experiments reveal that the first variant performs slightly better. Baldacci, Bartolini, and Mingozzi (2011) later added multiple ways of computing a dual bound and also used this bound for pricing. Other algorithms for the PDPTW (for goods transportation) and the related dial-a-ride problem (for passenger transportation) can be found in the surveys by Costa, Contardo, and Desaulniers (2019) and Berbeglia et al. (2007) or the book chapter of Vigo and Toth (2014).

MAPF considers a set of agents, each with a start location and end location, on a two-dimensional map. The problem aims to find a path for every agent from its start location to its end location such that the agents do not collide into each other and that the sum of end times is minimized. MAPF does not consider a set of orders but simply attempts to navigate every agent from its start location directly to its end location without vertex and edge collisions. The current state-of-the-art for exact MAPF are three competing tree search algorithms: (1) CBSH2-RCT (Li et al. 2021), a new variant of conflict-based search (CBS) (Sharon et al. 2015), which performs a high-level tree search and solves a low-level path finding problem at every node of the search tree, (2) Lazy CBS (Gange, Harabor, and Stuckey 2019), which adds conflict-driven clause learning from propositional satisfiability (SAT) and constraint programming to CBS, and (3) BCP-MAPF (Lam et al. 2022), a branch-and-cut-and-price algorithm with eleven classes of valid inequalities and several acceleration techniques.

The MAPD problem can be divided into a PDPTW component that constructs a sequence of requests for each agent and a MAPF component that finds a path for every agent to visit the locations of its assigned requests. The main difference between the PDPTW and MAPD is that the travel distances in the PDPTW is given as a look-up matrix, whereas in MAPD, they are computed as the solution to a MAPF problem, which could change due to collisions.

This division of the problem naturally suggests that a decomposition approach that divides the work to dedicated algorithms could be effective. Dantzig-Wolfe and Benders decomposition are effective at integer programming problems with substructures that can be solved quickly using specialized algorithms but are complicated by constraints that span across multiple substructures. A Dantzig-Wolfe approach often includes a branch-and-price algorithm, which dynamically builds the linear relaxation of the integer programming problem column-by-column. A Benders method correlates with a branch-and-cut algorithm, which analogously builds the linear relaxation row-by-row. Formal details on these techniques can be found in the references by Lübbecke and Desrosiers (2005), Desrosiers and Lübbecke (2010) and Wolsey (2021).

Dantzig-Wolfe decomposition can natively handle discrete variables in the master problem and subproblem. In contrast, classical Benders decomposition can only accommodate discrete variables in the master problem. This limitation can be addressed using several techniques. In logic-based Benders decomposition, an inference dual must be defined and manually analyzed to generate Benders cuts valid for one particular problem (Hooker and Ottosson 2003). Benders cuts can also be generated automatically for arbitrary problems by repurposing the irreducible inconsistent subsystem from integer programming (Codato and Fischetti 2006), conflict-driven clause learning from propositional satisfiability (SAT) (Lam and Van Hentenryck 2017) and lazy clause generation from constraint programming (Davies, Gange, and Stuckey 2017, Lam et al. 2020). These methods generate a Benders cut using a certificate of infeasibility or suboptimality in the discrete subproblem and are analogous to the derivation of classical Benders cuts using linear programming duality theory and the Farkas lemma.

Many suboptimal algorithms have been proposed for MAPD (e.g., Ma et al. 2017, Liu et al. 2019, Chen et al. 2021, Xu et al. 2022) but only four exact approaches are found in the literature, all of which present MAPD under a different name. Henkel, Abbenseth, and Toussaint (2019) solve MAPD without time windows under the name of Combined Task Allocation and Path Finding. They propose a simple extension of CBS to consider the task assignment but could only solve trivially small instances.

Ren, Rathinam, and Choset (2023) study Multi-Agent Combinatorial Path Finding, which is essentially MAPD but the task assignment step is modeled on the Multi-Traveling Salesman Problem, instead of the PDPTW, and therefore ignores the load, time window and pickup-delivery constraints of the PDPTW. They propose a tree search algorithm based on CBS that calls an external solver for the Traveling Salesman Problem. While their tree search algorithm is theoretically optimal, the experiments unfortunately executed the Lin-Kernighan-Helsgaun heuristic (Helsgaun 2017) for solving the Traveling Salesman Problems and hence voided all optimality guarantees.

A problem closely related to MAPD is studied under the name of routing of automated guided vehicles (AGVs). These problems often appear in manufacturing and container terminals. Despite the large body of work on routing AGVs, few papers consider the task assignment and conflict-free path planning problems together. The main differences between these problems and MAPD are that the objective usually minimizes delays and that the network is defined at a higher-level (e.g., the nodes and edges of the network represent junctions and segments of the rail/track, rather than square cells), whereas the map in MAPD inherits the two-dimensional grid environment from MAPF.

Desaulniers et al. (2003) propose an exact branch-and-cut-and-price algorithm for the AGV conflict-free routing problem. They solve the task assignment and path finding problems using one integrated graph, which makes for a more complex mathematical model. In comparison, our approach defines two distinct graphs for task assignment and path finding, which provide a richer set of variables to define constraints, branching rules, etc. We also exploit the grid layout of MAPD to develop additional reasoning techniques (e.g., valid inequalities).

Corréa, Langevin, and Rousseau (2007) design a hybrid exact approach based on logic-based Benders decomposition for conflict-free AGV routing. They use a constraint programming master problem to schedule the tasks and an integer programming subproblem for path finding. This method mitigates issues of the branch-and-cut-and-price approach by Desaulniers et al. (2003) related to relying on a warm-start solution found by a heuristic.

Other combinations of task assignment and MAPF have also appeared in one form or another. The same problem often appears under different names, as seen above, and make a literature search difficult. Liu et al. (2019) and Ren, Rathinam, and Choset (2023) describe some of these problems.

3 Problem Definition

The MAPD problem is defined on a two-dimensional rectangular map with a set of traversable locations $\mathcal{L} = \{(x_1, y_1), \ldots, (x_{|\mathcal{L}|}, y_{|\mathcal{L}|})\}$ given by their east-west and north-south coordinates. Non-traversable locations, called obstacles, are omitted from $\mathcal{L}$. A location $l_2 = (x_2, y_2) \in \mathcal{L}$ is a neighbor of location $l_1 = (x_1, y_1) \in \mathcal{L}$ in the

- north direction if $x_2 = x_1$ and $y_2 = y_1 - 1$,
- south direction if $x_2 = x_1$ and $y_2 = y_1 + 1$,
- west direction if $x_2 = x_1 - 1$ and $y_2 = y_1$, and
- east direction if $x_2 = x_1 + 1$ and $y_2 = y_1$.

Under this definition, the north-west corner of the map is the origin $(0, 0)$.

Let $T \in \mathbb{Z}_+$ be the number of timesteps in the planning period and $\mathcal{T} = \{0, \ldots, T - 1\}$ be the set of discrete timesteps. The problem is defined on a time-expanded directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{L} \times \mathcal{T}$ is the set of vertices and $\mathcal{E} = \{(((x_1, y_1), t_1), ((x_2, y_2), t_2)) \in \mathcal{V} \times \mathcal{V} : |x_2 - x_1| + |y_2 - y_1| \leq 1 \wedge t_2 = t_1 + 1\}$ is the set of edges. A vertex $v \in \mathcal{V}$ is a location-timestep pair. An edge $e \in \mathcal{E}$ represents a movement from a location at some timestep to a neighbor location (a move action) or a movement to the same location (a wait action) in the next timestep. The reverse $e' = ((l_2, t_1), (l_1, t_1 + 1))$ of an edge $e = ((l_1, t_1), (l_2, t_1 + 1))$ is a movement in the opposite direction at the same timestep.

Let $\mathcal{A}$ be the set of agents. Every agent $a \in \mathcal{A}$ has a start location $L_a^+ \in \mathcal{L}$ and an end location $L_a^- \in \mathcal{L}$. While the start location and end location of an agent can be identical, all start locations are unique and all end locations are unique. A path p of length $k \in \{1, \ldots, T\}$ for agent a is a sequence of k locations $(l_0, \ldots, l_{k-1})$ such that $l_0 = L_a^+$, $l_{k-1} = L_a^-$ and $((l_t, t), (l_{t+1}, t + 1)) \in \mathcal{E}$ for all $t \in \{0, \ldots, k - 2\}$. For convenience, define $l_t = L_a^-$ for all $t \in \{k, \ldots, T - 1\}$ because the agent remains at its end location after the path ends. Path p visits the vertex (l_t, t) for all $t \in \mathcal{T}$ and traverses the edge $((l_t, t), (l_{t+1}, t + 1))$ for all $t \in \{0, \ldots, T - 2\}$. Path p has a cost $c_p = k - 1$ equal to the number of actions required to reach the end location and wait there indefinitely.

Let $O \in \mathbb{Z}_+$ be the number of orders and let $\mathcal{O} = \{(r_1^\uparrow, r_1^\downarrow), \ldots, (r_O^\uparrow, r_O^\downarrow)\}$ be the set of orders, where an order is a pair of a pickup request and a delivery request. Define $\mathcal{R}^\uparrow = \{r_1^\uparrow, \ldots, r_O^\uparrow\}$ and $\mathcal{R}^\downarrow = \{r_1^\downarrow, \ldots, r_O^\downarrow\}$ as the set of pickup requests and delivery requests respectively. Every request $r \in \mathcal{R}^\uparrow \cup \mathcal{R}^\downarrow$ is located at $L_r \in \mathcal{L}$ and must occur between $\underline{T}_r \in \mathcal{T}$ and $\bar{T}_r \in \mathcal{T}$ inclusive, where $\underline{T}_r \leq \bar{T}_r$.

The MAPD problem assigns every order to an agent and assigns a path to every agent such that the path visits the locations of the pickup and delivery requests of all orders assigned to the agent. If an order $(r^\uparrow, r^\downarrow) \in \mathcal{O}$ is assigned to an agent, then the path assigned to the agent must visit the vertices $(L_{r^\uparrow}, t_{r^\uparrow}) \in \mathcal{V}$ and $(L_{r^\downarrow}, t_{r^\downarrow}) \in \mathcal{V}$ at some time $t_{r^\uparrow} \in \{\underline{T}_{r^\uparrow}, \ldots, \bar{T}_{r^\uparrow}\}$ and $t_{r^\downarrow} \in \{\underline{T}_{r^\downarrow}, \ldots, \bar{T}_{r^\downarrow}\}$, where $t_{r^\uparrow} \leq t_{r^\downarrow}$. At any given time, an agent can undertake at most one order, i.e., $t_{r_1^\downarrow} \leq t_{r_2^\uparrow}$ or $t_{r_2^\downarrow} \leq t_{r_1^\uparrow}$ for any two distinct orders $(r_1^\uparrow, r_1^\downarrow), (r_2^\uparrow, r_2^\downarrow) \in \mathcal{O}$ assigned to the agent.

The paths assigned to the agents must be free of vertex collisions and edge collisions, i.e., if an agent takes the path $p_1 = (l_0^{p_1}, \ldots, l_{k_1-1}^{p_1})$ and a different agent takes the path $p_2 = (l_0^{p_2}, \ldots, l_{k_2-1}^{p_2})$, the conditions $l_t^{p_1} \neq l_t^{p_2}$ and $l_t^{p_1} \neq l_{t+1}^{p_2} \vee l_t^{p_2} \neq l_{t+1}^{p_1}$ must hold for all $t \in \mathcal{T}$.

A feasible solution consists of paths that satisfy all these conditions. An optimal solution is a feasible solution that minimizes the sum of path costs.

4 BCP-MAPD

This section presents BCP-MAPD, the first of two branch-and-cut-and-price algorithms for exact MAPD. It consists of four main components:

- The (restricted) master problem is the linear relaxation of an integer programming problem based on a set partitioning formulation. Given a set of pairs of a request sequence and a path for every agent, the master problem chooses elements from these sets to assemble a valid MAPD solution. For every agent, it selects a fractionally-optimal subset of request sequences and paths from the huge but incomplete set such that every request is completed exactly once and that the selected paths together are free of vertex conflicts and edge conflicts. The master problem, being a linear programming problem, is explicitly allowed to select multiple paths for each agent on the condition that the proportions of the request sequences and paths selected for each agent sum to 100%.

- The pricers solve the pricing problem of every agent, which is a two-level resource-constrained shortest path problem. Solutions to a pricing problem correspond to new request sequences and paths that could potentially appear in a future lower-cost solution of the master problem and therefore are added to master problem, allowing it to select these in later iterations.
- The separators resolve conflicts in solutions to the master problem. Every solution must be checked by the separators to ensure that they are free of several types of conflicts. If conflicts occur, the separators add constraints to the master problem to prohibit certain combinations of request sequences and paths.
- The branching rules build the branch-and-bound tree to progressively remove the fractionalities in the master problem. One branching rule makes guesses as to whether an agent does or does not take an edge, incrementally forcing the master problem to select fewer and fewer paths for each agent until eventually it finds a solution that selects one path with 100% proportion for each agent. A complete exploration of the search tree will obtain an optimal solution if one exists.

Let the navigation graph $\mathcal{G}^{\text{nav}} = (\mathcal{V}^{\text{nav}}, \mathcal{E}^{\text{nav}}) = \mathcal{G}$ explicitly denote the time-expanded graph $\mathcal{G}$. Define the sequencing graph $\mathcal{G}^{\text{seq}} = (\mathcal{V}^{\text{seq}}, \mathcal{E}^{\text{seq}})$ with vertices $\mathcal{V}^{\text{seq}} = \mathcal{R}^{\uparrow} \cup \mathcal{R}^{\downarrow} \cup \{\top, \bot\}$ where $\top$ and $\bot$ are source and sink vertices representing the start and end location of an agent. The edges

$$\begin{aligned}\mathcal{E}^{\text{seq}} = &\{(\top, \bot)\} \cup \\ &\{(\top, r^{\uparrow}) : r^{\uparrow} \in \mathcal{R}^{\uparrow}\} \cup \\ &\{(r^{\uparrow}, r^{\downarrow}) : (r^{\uparrow}, r^{\downarrow}) \in \mathcal{O}\} \cup \\ &\{(r^{\downarrow}, r^{\uparrow}) : r^{\downarrow} \in \mathcal{R}^{\downarrow}, r^{\uparrow} \in \mathcal{R}^{\uparrow}, (r^{\uparrow}, r^{\downarrow}) \notin \mathcal{O}\} \cup \\ &\{(r^{\downarrow}, \bot) : r^{\downarrow} \in \mathcal{R}^{\downarrow}\}\end{aligned}$$

are partitioned into five subsets, which respectively represent movements from the start location to the end location without completing any order, from the start location to a pickup, from a pickup to its corresponding delivery, from a delivery to a different pickup, and from a delivery to the end location. The main idea behind BCP-MAPD is to simultaneously search for paths on the sequencing graph and the navigation graph.

4.1 The Master Problem

Define a request sequence $s = (\top, r_1, \dots, r_n, \bot)$ as a path on $\mathcal{G}^{\text{seq}}$ from the source $\top$ to the sink $\bot$, where $r_1, \dots, r_n \in \mathcal{R}^{\uparrow} \cup \mathcal{R}^{\downarrow}$ are the pickup and delivery requests completed in s. The requests $r_1, \dots, r_n$ are not necessarily unique, i.e., a request can be completed more than once within a sequence.

Every request sequence s is associated with an agent $a \in \mathcal{A}$ and a path $p = (l_0, \dots, l_{k-1})$ of length k on $\mathcal{G}^{\text{nav}}$ that navigates the agent to the locations of the requests $r_1, \dots, r_n$, i.e., $l_0 = L_a^+, l_{k-1} = L_a^-$ and there exists $t_1 \leq t_2 \leq \ldots \leq t_n$ with $l_{t_i} = L_{r_i}$ and $\underline{T}_{r_i} \leq t_i \leq \bar{T}_{r_i}$ for all $i \in \{1 \dots, n\}$. Note that a path can pass through the location of a request but not necessarily complete it.

The master problem is a linear program that selects a subset of sequence-path pairs for every agent such that the proportions of all sequence-path pairs selected for an agent sum to 100%, every request is completed with 100% proportion across all selected request sequences, the paths across all agents are fractionally free of conflicts, and the total cost is minimized.

For all $a \in \mathcal{A}$, let $\Lambda_a = \{(s_1^a, p_1^a), \dots, (s_{|\Lambda_a|}^a, p_{|\Lambda_a|}^a)\}$ be the large set of sequence-path pairs from which a subset is selected, let $\lambda_{a,s,p} \in [0, 1]$ be a decision variable representing the proportion of selecting $(s, p) \in \Lambda_a$, and let $\alpha_s^r \in \mathbb{Z}_+$ be a constant indicating the number of times that request $r \in \mathcal{R}^{\uparrow} \cup \mathcal{R}^{\downarrow}$ appears in sequence s. The master problem begins as the linear program:

$$\min \sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} c_p \lambda_{a,s,p} \tag{1a}$$

subject to

$$\sum_{(s,p) \in \Lambda_a} \lambda_{a,s,p} = 1 \qquad \forall a \in \mathcal{A}, \tag{1b}$$

$$\sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} \alpha_s^r \lambda_{a,s,p} = 1 \qquad \forall r \in \mathcal{R}^{\uparrow}, \tag{1c}$$

$$\lambda_{a,s,p} \geq 0 \qquad \forall a \in \mathcal{A}, (s, p) \in \Lambda_a. \tag{1d}$$

Objective Function (1a) minimizes the total cost of the selected paths. Constraint (1b) requires the proportions of the sequence-path pairs selected for every agent to sum to 100%. Constraint (1c) requires every pickup request to be completed with 100% proportion across all agents. By the definition of $\mathcal{G}^{\text{seq}}$, every delivery request must be completed immediately after its corresponding pickup request and therefore the master problem does not need to consider delivery requests. Constraint (1d) are the non-negativity constraints standard in linear programming. Constraints (1b) and (1d) together ensure that $\lambda_{a,s,p} \in [0, 1]$.

Constraints prohibiting vertex conflicts and edge conflicts are initially omitted and added dynamically as necessary. BCP-MAPD incrementally builds the sets Λ_a and the vertex conflict and edge conflict constraints.

4.2 Resolving Conflicts

Whenever a feasible solution to the master problem is found, separators must check it for violation of the vertex conflict and edge conflict conditions. If so, a constraint is added to the master problem to resolve the conflict, forcing it to choose different sequence-path pairs.

Define the constant $\alpha_p^e \in \{0, 1\}$ to count the number of times that the edge $e \in \mathcal{E}^{\text{nav}}$ appears in path p. Let $\mathcal{B}$ be the set of conflict constraints. Every conflict constraint $b \in \mathcal{B}$ can be defined by the constants $\beta_b^{a,e} \in \mathbb{Z}_+$, where $a \in \mathcal{A}, e \in \mathcal{E}^{\text{nav}}$, and $\beta_b \in \mathbb{Z}_+$ in the form

$$\sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} \left(\sum_{e \in \mathcal{E}^{\text{nav}}} \beta_b^{a,e} \alpha_p^e \right) \lambda_{a,s,p} \leq \beta_b. \tag{2}$$

Constraints in the form of Constraint (2) are described as *robust* because they only impact the pricing problem in the objective function (Fukasawa et al. 2006).

Constraints enforcing vertex conflicts, edge conflicts and several other types of conflicts are described below. All of these have previously appeared in BCP-MAPF (Lam et al. 2022).

4.2.1 Edge Conflicts

An edge conflict occurs if an edge $e \in \mathcal{E}^{\text{nav}}$ and its reverse e' together are used more than once (with more than 100% proportion). The number x_e of times that an edge e or its reverse e' are traversed by all agents can be computed as

$$x_e = \sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} (\alpha_p^e + \alpha_p^{e'}) \lambda_{a,s,p}.$$

An edge conflict occurs at $e \in \mathcal{E}^{\text{nav}}$ whenever $x_e > 1$ and the constraint

$$\sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} (\alpha_p^e + \alpha_p^{e'}) \lambda_{a,s,p} \leq 1 \tag{3}$$

is added to the master problem to resolve this edge conflict.

4.2.2 Vertex Conflicts

Vertex conflicts can be resolved similarly by recognizing that a vertex conflict occurs whenever the five incoming edges to a vertex are used more than once. For any vertex $v = (l, t) = ((x, y), t) \in \mathcal{V}^{\text{nav}}$, define

$$\begin{aligned} \alpha_p^v = &\alpha_p^{(((x-1,y),t-1),(l,t)))} + \alpha_p^{(((x+1,y),t-1),(l,t)))} + \\ &\alpha_p^{(((x,y-1),t-1),(l,t)))} + \alpha_p^{(((x,y+1),t-1),(l,t)))} + \\ &\alpha_p^{((l,t-1),(l,t)))} \end{aligned}$$

as the number of times that path p visits vertex v. The number x_v of times that vertex v is visited by all agents can be calculated as

$$x_v = \sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} \alpha_p^v \lambda_{a,s,p}.$$

A vertex conflict occurs at $v \in \mathcal{V}^{\text{nav}}$ whenever $x_v > 1$. The constraint

$$\sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} \alpha_p^v \lambda_{a,s,p} \leq 1 \tag{4}$$

is then added to the master problem to resolve this vertex conflict.

4.2.3 Other Types of Conflicts

Including constraints for removing vertex conflicts and edge conflicts is sufficient for correctly solving the MAPD problem. However, typical in integer programming approaches, the master problem can be tightened using other constraints that provide additional reasoning.

Several families of robust valid inequalities for MAPF also implemented (Lam et al. 2022, Lam and Le Bodic 2020). Constraint (3) for resolving edge conflicts are lifted to the wait edge conflict constraints. Rectangle, wait corridor and wait two-edge conflicts are also implemented. These were previously shown to have the greatest impact on the solve time for MAPF. Due to the complexity of the pricing code, non-robust cuts for MAPF are not considered.

The subset row inequalities for VRPs (Jepsen et al. 2008), which reason about a set packing relaxation, are also implemented. Note that while many families of cuts have been developed for VRPs, many are theoretically (Letchford and Salazar-González 2006) and/or experimentally (Costa, Contardo, and Desaulniers 2019) shown to be ineffective within a branch-and-cut-and-price setting due to the tight Dantzig-Wolfe reformulation and therefore are not considered here.

4.3 Generating Sequences and Paths

In general, the set Λ_a of sequence-path pairs for any agent $a \in \mathcal{A}$ is exponential in the instance size. Therefore, only a small but sufficient number of elements are generated on-demand. They are generated by solving the pricing problem for every agent, whose solutions correspond to new sequence-path pairs that are added to the master problem.

The pricing problem of every agent is a two-level shortest path problem. A high-level resource-constrained shortest path problem finds a request sequence by searching for a path on $\mathcal{G}^{\text{seq}}$ from $\top \in \mathcal{V}^{\text{seq}}$ to $\bot \in \mathcal{V}^{\text{seq}}$, which respectively represent the start and end location of agent a. It considers reduced cost and time resources. The high-level problem is solved by a labeling algorithm commonly seen in the VRP literature. The labeling algorithm starts with a partial sequence starting and ending at the source $\top$ and iteratively extends it to form longer partial sequences until eventually reaching $\bot$.

When extending a high-level partial sequence along an edge $(i, j) \in \mathcal{E}^{\text{seq}}$, a low-level shortest path problem is solved using an A* algorithm to find a path on $\mathcal{G}^{\text{nav}}$ from $(L_i, t_i) \in \mathcal{V}^{\text{nav}}$ to $(L_j, t_j) \in \mathcal{V}^{\text{nav}}$ at some time $t_j \in \{\underline{T}_j, \dots, \bar{T}_j\}$, where the source and sink locations $L_\top := L_a^+$ and $L_\bot := L_a^-$ are the start and end location of agent a. This path segment navigates the agent from its current location to the location of the next request or its end location. There are no resource constraints in the low-level problem but there are reduced cost and time resources in addition to a third resource for keeping track of the extra penalties in the reduced cost above the true cost of the path. When solving the low-level shortest path problem, every Pareto-optimal path across the three resources corresponds to a new partial sequence in the high-level problem. (See Section 4.3.7 for more detail.) A Pareto frontier of low-level paths is necessary to guarantee optimality of the pricing problem because reduced cost and time can be traded-off in the high-level problem.

In the minimization master problem, the reduced cost of a sequence-path pair is a bound on the change in the objective value of the master problem if this sequence-path pair is added. Therefore, adding sequence-path pairs with negative reduced cost could potentially lead to lower-cost feasible solutions. A global minimum of the master problem is attained when all solutions to the pricing problems have non-negative reduced cost (Lübbecke and Desrosiers 2005).

The remainder of this section discusses several intricate details that must be precisely modeled to correctly define and solve the pricing problems.

4.3.1 Reduced Cost Function

While any sequence-path pair with negative reduced cost corresponds to a new column in the master problem, the pricing problem is often posed as a minimization problem to find a column with the most negative reduced cost. The intuition behind this greedy heuristic, which may not be entirely true in practice, is that it can minimize the master problem as quickly as possible.

Let $\pi_a, \rho_r \in \mathbb{R}, \sigma_b \in \mathbb{R}_-$ be the dual variables of Constraints (1b), (1c) and (2) respectively. When solving the pricing problem for agent $a \in \mathcal{A}$, a sequence-path pair (s, p) with sequence $s = (\top, r_1, \dots, r_n, \bot)$ and path $p = (l_0, \dots, l_{k-1})$ has reduced cost

$$\bar{c}_{(s,p)} = c_p - \pi_a - \sum_{i=1}^{n} \rho_{r_i} - \sum_{t=0}^{T-2} \sum_{b \in \mathcal{B}} \beta_b^{a,((l_t,t),(l_{t+1},t+1))} \sigma_b.$$

4.3.2 Encoding Reduced Costs on the Networks

The reduced cost function can be encoded as edge costs in the two-level shortest path problem. In the high-level shortest path problem on $\mathcal{G}^{\text{seq}}$, the initial partial sequence has cost $-\pi_a$, every edge incoming to a pickup request $r \in \mathcal{R}^{\uparrow}$ has cost $-\rho_r$ and the other edges have 0 cost. In the low-level shortest path problem on $\mathcal{G}^{\text{nav}}$, every edge $e \in \mathcal{E}^{\text{nav}}$ has a default cost of 1 due to the path cost $c_p = k-1$, plus an extra cost $-\sum_{b\in\mathcal{B}} \beta_b^{a,e}\sigma_b$.

Recall from Section 3 that an agent a taking a path $p = (l_0, \dots, l_{k-1})$ of length k will traverse the edges $((l_t, t), (l_{t+1}, t+1))$ for all $t \in \{0, \dots, k-2\}$, where $l_0 = L_a^+$ and $l_{k-1} = L_a^-$, and then the edges $((L_a^-, t), (L_a^-, t+1))$ for all $t \in \{k-1, \dots, T-2\}$ because the agent waits at its end location L_a^- indefinitely.

When finding a shortest path to the end $(L_a^-, k-1) \in \mathcal{V}^{\text{nav}}$, all reduced costs on the later edges $((L_a^-, t), (L_a^-, t+1)) \in \mathcal{E}^{\text{nav}}$, where $t \in \{k-1, \dots, T-2\}$, will be ignored because they occur after the path terminates at time $k-1$. (These reduced costs can arise from other agents attempting to cross L_a^- after time k, for instance.)

Whenever a partial sequence $s = (\top, r_1, \dots, r_n)$ ending at the request r_n is extended to the sink $\perp$, the low-level problem needs to find a shortest path on a modified graph $\mathcal{G}^{\text{navend}} = (\mathcal{V}^{\text{navend}}, \mathcal{E}^{\text{navend}})$ to account for the reduced costs on waiting at the end location after time $k-1$. The vertices $\mathcal{V}^{\text{navend}} = \mathcal{V}^{\text{nav}} \cup \{\triangle\}$ includes a dummy sink vertex $\triangle$ and the edges $\mathcal{E}^{\text{navend}} = \mathcal{E}^{\text{nav}} \cup \{((L_a^-, t), \triangle) : t \in \mathcal{T}\}$ contains additional edges representing the agent completing its path. Every edge $((L_a^-, t), \triangle) \in \mathcal{E}^{\text{navend}}$ is given a cost $-\sum_{t'=t}^{T-2} \sum_{b\in\mathcal{B}} \beta_b^{a,((L_a^-,t'),(L_a^-,t'+1))}\sigma_b$. When extending a partial sequence to the sink $\perp$, using the modified graph $\mathcal{G}^{\text{navend}}$ will correctly accumulate edge costs for waiting at the end location L_a^- after time $k-1$. When extending a partial sequence to any regular request vertex $r \in \mathcal{R}^{\uparrow} \cup \mathcal{R}^{\downarrow}$, the original graph $\mathcal{G}^{\text{nav}}$ is used.

4.3.3 Completion Bounds in the Navigation Level

When extending a partial sequence along an edge $(i, j) \in \mathcal{E}^{\text{seq}}$ and solving for a path from $(L_i, t_i) \in \mathcal{V}^{\text{nav}}$ to $(L_j, t_j) \in \mathcal{V}^{\text{nav}}$, the A* algorithm relies on a lower bound function on the cost-to-go, called the *heuristic*, to estimate the lowest possible cost from any vertex $v \in \mathcal{V}^{\text{nav}}$ to (L_j, t_j).

The heuristic function h is computed as follows. First, an A* algorithm is run on a non-time-expanded graph whose vertices are the locations $\mathcal{L}$ to pre-compute the minimum number of steps to L_j from all locations $l \in \mathcal{L}$. This A* algorithm is run using the Manhattan distance as its heuristic. The costs resulting from this computation is then used as the heuristic for the A* search on the time-expanded network $\mathcal{G}^{\text{nav}}$. When extended to the sink $\triangle$ on the modified graph $\mathcal{G}^{\text{navend}}$, the minimum sum-of-reduced-costs on the wait edges is also included.

4.3.4 Reduced Costs on Outgoing Edges at Request Locations

Consider a request i with time window $[5, 9]$ and a partial sequence ending at i being extended to another request. Figure 1 illustrates the extension of a corresponding partial path ending at location L_i to a neighbor location l at various timesteps. All outgoing edges of vertices $(L_i, 5)$ and $(L_i, 6)$ have reduced cost 10 and all outgoing edges of vertices $(L_i, 7)$, $(L_i, 8)$ and $(L_i, 9)$ have reduced cost 1.

If searching for a lowest cost path to location L_i, the path would terminate at vertex $(L_i, 5)$ at time 5 because any path arriving after time 5 would cost more. However, the myopic nature of this extension will not see that a future extension from $(L_i, 5)$ onward would cost more than extensions from $(L_j, 7)$.

For every vertex $i \in \mathcal{V}^{\text{seq}}$, define a set $\mathcal{I}_i = \{(\underline{t}_1, \bar{t}_1), \dots, (\underline{t}_{|\mathcal{I}_i|}, \bar{t}_{|\mathcal{I}_i|})\} \subset \mathcal{T} \times \mathcal{T}$ whose elements $(\underline{t}, \bar{t}) \in \mathcal{I}_i$ represent maximal time ranges such that all outgoing edges from $(L_i, t) \in \mathcal{V}^{\text{nav}}$, $t \in \{\underline{t}, \dots, \bar{t}\}$, have identical reduced costs. In the example in Figure 1, $\mathcal{I}_i = \{(5, 6), (7, 9)\}$. The sequencing problem is forced to find paths to L_i during every time range $[\underline{t}, \bar{t}] \in \mathcal{I}_i$, instead of just its time window $[\underline{T}_i, \bar{T}_i]$. This ensures that future extensions are not missed. The search will now find two paths ending at $(L_i, 5)$ and $(L_i, 7)$.

4.3.5 Dominance Rules in the Sequencing Level

Labeling algorithms for resource-constrained shortest path problems rely on dominance rules to prevent the exploration of every feasible sequence. For a partial sequence s ending at a vertex i, let $\bar{c}_s$ and τ_s respectively be the accumulation of reduced costs and the arrival time. Given two partial sequences s_1 and s_2 ending at a common vertex i with resource consumptions $\bar{c}_{s_1}, \tau_{s_1}$ and $\bar{c}_{s_2}, \tau_{s_2}$, the sequence s_1 *dominates* s_2 and s_2 can be discarded from further consideration if

$$\tau_{s_1} \leq \tau_{s_2} \tag{5a}$$

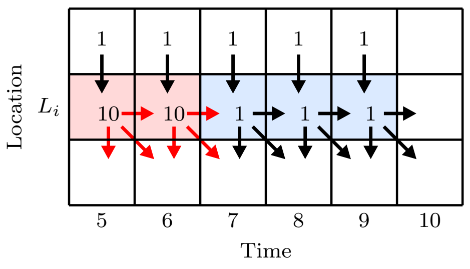

Figure 1: The vertex $i \in \mathcal{V}^{\text{seq}}$ is associated with two time ranges $[5, 6]$ and $[7, 9]$. Within each of these time ranges, the location L_i has identical reduced costs on all outgoing edges on the navigation graph.

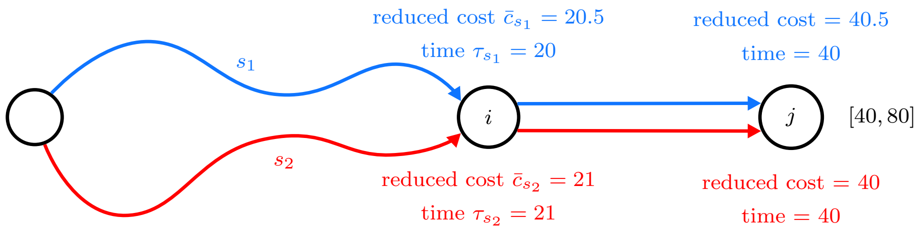

Figure 2: Two sequences demonstrating that the standard dominance rules are not valid for the navigation problem.

and

$$\bar{c}_{s_1} + \sum_{t=\tau_{s_1}}^{\tau_{s_2}-1} \left(1 - \sum_{b \in \mathcal{B}} \beta_b^{a,((L_i,t),(L_i,t+1))} \sigma_b \right) \leq \bar{c}_{s_2}. \tag{5b}$$

Condition (5a) is the usual time condition, which states that s_1 can dominate s_2 if s_1 arrives before or at the same time as s_2. Condition (5b) says that s_1 can dominate s_2 if arriving at L_i at the arrival time τ_{s_1} of s_1 and then waiting until the arrival time τ_{s_2} of s_2 gives a cheaper partial sequence than s_2. A consequence of this condition is that any extension of s_2 concatenated on s_1 and then waiting from time τ_{s_1} to τ_{s_2} is cheaper than the same extension of s_2.

Figure 2 shows a counterexample of why the sum in Condition (5b) is necessary. Two sequences s_1 and s_2 arrive at vertex i respectively with reduced cost $\bar{c}_{s_1} = 20.5$ and time $\tau_{s_1} = 20$, and reduced cost $\bar{c}_{s_2} = 21$ and time $\tau_{s_2} = 21$. According to the classical dominance rules $\bar{c}_{s_1} \leq \bar{c}_{s_2}$ and $\tau_{s_1} \leq \tau_{s_2}$, s_1 dominates s_2. Consider extending these two sequences to vertex j whose time window is $[40, 80]$. Suppose that j is 19 locations away. The sequence s_1 arrives at j at time $20 + 19 = 39$ but must wait 1 timestep until the time window opens, resulting in a reduced cost of $20.5 + 19 + 1 = 40.5$ and a time of $20 + 19 + 1 = 40$. The sequence s_2 arrives at j at time $21 + 19 = 40$, when the time window is already open. According to the usual dominance rules, now the extension of s_2 dominates the extension of s_1. Condition (5b) states that a partial sequence can only dominate another if extending it to the time of the other gives a lower or identical reduced cost. Using this condition, s_1 does not dominate s_2 at vertex i and hence both extensions to vertex j are explored.

Condition (5b) can be strengthened by using the reduced cost of the true shortest path from (L_i, τ_{s_1}) to (L_i, τ_{s_2}) but computing this path cost is likely to be prohibitively expensive because the dominance check is performed for a huge number of partial sequences. Therefore, the weaker Condition (5b) is used instead. For the vast majority of cases, waiting is likely to be optimal anyway.

4.3.6 Preventing Subtours

Labeling algorithms for the pricing problem in VRPs often include one binary resource for every request to indicate whether the request has been completed (Feillet et al. 2004). These extra resources are used to ensure that a vertex is visited at most once along a path. Røpke and Cordeau (2009) experimented with including and excluding these resources for the PDPTW and found that including them slightly improved run time. In contrast, preliminary experiments for MAPD conclusively demonstrate that they substantially worsen performance and hence are not considered any further. In the absence of these resources, a sequence can visit a request vertex multiple times but it will be infeasible in an integer solution

```
1  Function Pricer($(\mathcal{I}_i)_{i\in\mathcal{V}^{seq}}, \pi_a, (\rho_j)_{j\in\mathcal{R}^\uparrow}, (\sigma_b)_{b\in\mathcal{B}}$):
       input  : the time ranges of all vertices $(\mathcal{I}_i)_{i\in\mathcal{V}^{\text{seq}}}$, the dual solution $\pi_a$ of Constraint (1b) for agent $a$, the
                dual solutions $(\rho_j)_{j\in\mathcal{R}^\uparrow}$ of Constraint (1c), the dual solutions $(\sigma_b)_{b\in\mathcal{B}}$ of Constraint (2)
       output : a set of sequence-path pairs with negative reduced cost
2      forall $i \in \mathcal{V}^{\text{seq}}$ do
3          $labels_i \leftarrow$ NewSet()
4      $open \leftarrow$ NewPriorityQueue()
5      $open$.Add($((\top), ((L_\top, 0)), -\pi_a, 0)$)
6      while $\neg open$.IsEmpty() do
7          $((\top, r_1, \ldots, r_n), ((l_0, 0), \ldots, (l_t, t)), \bar{c}, t) \leftarrow open$.Pop()
8          forall $j \in \mathcal{V}^{\text{seq}} : (r_n, j) \in \mathcal{E}^{\text{seq}}$ do
9              $(\top, r'_1, \ldots, r'_n) \leftarrow (\top, r_1, \ldots, r_n, j)$
10             $paths \leftarrow$ NewSet()
11             forall $(\underline{t}_j, \bar{t}_j) \in \mathcal{I}_j$ do
12                 $paths \leftarrow paths \cup$ A*($((l_0, 0), \ldots, (l_t, t)), \bar{c}, t, L_j, [\underline{t}_j, \bar{t}_j], (\sigma_b)_{b\in\mathcal{B}}$)
13             forall $(((l_0, 0), \ldots, (l_{t'}, t')), \bar{c}', t') \in paths$ do
14                 if $j \in \mathcal{R}^\uparrow$ then
15                     $\bar{c}' \leftarrow \bar{c}' - \rho_j$
16                 if $\neg$Dominated($(\bar{c}', t'), labels_j$) then
17                     $labels_j \leftarrow labels_j \cup \{((\top, r'_1, \ldots, r'_n), ((l_0, 0), \ldots, (l_{t'}, t')), \bar{c}', t')\}$
18                     $open$.Push($((\top, r'_1, \ldots, r'_n), ((l_0, 0), \ldots, (l_{t'}, t')), \bar{c}', t')$)
19     return $\{((\top, r_1, \ldots, r_n, \bot), ((l_0, 0), \ldots, (l_t, t)), \bar{c}, t) \in labels_\bot : \bar{c} < 0\}$
```

Algorithm 1: The algorithm for solving the two-level pricing problem.

due to Constraint (1c) and therefore omitting these request resources is valid. If the time windows are tight and travel distances are relatively long, the time window constraints also serve to eliminate subtours.

4.3.7 Resource for Penalties in the Navigation Level

It is well-established in the VRP literature that the pricing problem considers reduced cost and time resources. This remains true in the sequencing level of the pricing problem of MAPD. However, one more resource is needed in the navigation level due to the time expansion.

In the high-level sequencing problem, every vertex corresponds to a request and the dominance check can use Condition (5b) on partial sequences ending at the same vertex but have different arrival times. In the low-level navigation problem, the vertices are time-expanded and therefore this dominance rule is harder to implement. Instead, a *dual penalties* resource representing the accumulation of dual values (i.e., the difference between the reduced cost and time, after accounting for π_a) is necessary.

As an example, refer to the two sequences s_1 and s_2 ending at vertex i in Figure 2. If the low-level shortest path problem searches for paths ending at L_i with only the reduced cost and time resources, the path of s_1 is Pareto-optimal and hence the path of s_2 will not be found. This issue can be corrected by considering one more resource z for the penalties in the reduced cost corresponding to the dual values. Sequence s_1 has accumulated dual penalties of $z_{s_1} = 0.5$ whereas s_2 has received $z_{s_2} = 0$ dual penalties. Using the standard dominance rules on these three resources in the low-level shortest path problem ensures that both paths will be found.

4.3.8 Pseudocode

Algorithm 1 presents the pricer. A *label* is a 4-tuple that contains a partial sequence, the corresponding partial path, the accumulation of reduced costs and the arrival time. Lines 2 and 3 initialize a set of labels for every vertex to store the partial sequences ending at the vertex. Line 4 creates a priority queue of labels for future processing. This priority queue prioritizes labels with lower reduced cost. Line 5 creates the root label corresponding to the partial sequence-path pair starting and ending at $\top \in \mathcal{V}^{\text{seq}}$ and $(L_\top, 0) \in \mathcal{V}^{\text{nav}}$ with reduced cost $-\pi_a$ and arrival time 0. Lines 6 and 7 get a label out of the priority queue for processing. Line 8 iterates over every edge outgoing from the current request r_n. Line 9 extends the partial sequence. Line 10 creates a set to store the extensions of the partial path. Lines 11 and 12 extend the partial path $((l_0, 0), \ldots, (l_t, t))$ to (L_j, t') at all Pareto-optimal times $t' \in [\underline{t}_j, \bar{t}_j]$ within every time range $(\underline{t}_j, \bar{t}_j) \in \mathcal{I}_j$. Line 13 iterates over the new paths. If the new path ends at a pickup request (Line 14), Line 15 subtracts the dual solution of Constraint (1c) from the reduced cost. If the new label is not dominated (Line 16), Line 17 stores it in the set of labels for future dominance checks and Line 18 adds it to the priority queue. Line 19 returns the sequence-path pairs with negative reduced cost. These will be added to the master problem.

```
1  Function A*($((l_0,0),\ldots,(l_t,t)), \bar{c}, t, L_j, [\underline{t}_j, \bar{t}_j], (\sigma_b)_{b\in\mathcal{B}}$):
       input  : a partial path $((l_0,0),\ldots,(l_t,t))$, its reduced cost $\bar{c}$, its time $t$, the destination location $L_j$, the
                destination time range $[\underline{t}_j, \bar{t}_j]$, the dual solutions $(\sigma_b)_{b\in\mathcal{B}}$ of Constraint (2)
       output : a set of Pareto-optimal paths from $(L_a^{+}, 0)$ to $(L_j, t_j)$ where $t_j \in [\underline{t}_j, \bar{t}_j]$
2      $paths \leftarrow$ NewSet()
3      $\bar{t} \leftarrow \bar{t}_j$
4      $\bar{z} \leftarrow \infty$
5      $closed \leftarrow$ NewMap()
6      $closed[(l_t,t)] \leftarrow (((l_0,0),\ldots,(l_t,t)), \bar{c}, \bar{c} + h(l_t, L_j), 0)$
7      $open \leftarrow$ NewPriorityQueue()
8      $open$.Add($(l_t,t)$)
9      while $\neg open$.IsEmpty() do
10         $(l_t,t) \leftarrow open$.Pop()
11         $(((l_0,0),\ldots,(l_t,t)), g, f, z) \leftarrow closed[(l_t,t)]$
12         if $t \le \bar{t} \wedge z \le \bar{z}$ then
13             if $l_t = L_j \wedge t \ge \underline{t}_j$ then
14                 $\bar{t} \leftarrow t$
15                 $\bar{z} \leftarrow z$
16                 $paths \leftarrow paths \cup \{(((l_0,0),\ldots,(l_t,t)), g, t)\}$
17             forall $(l_{t+1}, t+1) \in \mathcal{V}^{\text{nav}} : ((l_t,t),(l_{t+1},t+1)) \in \mathcal{E}^{\text{nav}}$ do
18                 $g' \leftarrow g + 1 - \sum_{b\in\mathcal{B}} \beta_b^{a,((l_t,t),(l_{t+1},t+1))} \sigma_b$
19                 $h' \leftarrow h(l_{t+1}, L_j)$
20                 $f' \leftarrow g' + h'$
21                 $z' \leftarrow z - \sum_{b\in\mathcal{B}} \beta_b^{a,((l_t,t),(l_{t+1},t+1))} \sigma_b$
22                 if $\neg closed$.Contains($(l_{t+1}, t+1)$) then
23                     $closed[(l_{t+1},t+1)] \leftarrow (((l_0,0),\ldots,(l_t,t),(l_{t+1},t+1)), g', f', z')$
24                     $open$.Push($(l_{t+1},t+1), f'$)
25                 else if $closed[(l_{t+1},t+1)].f > f'$ then
26                     $closed[(l_{t+1},t+1)] \leftarrow (((l_0,0),\ldots,(l_t,t),(l_{t+1},t+1)), g', f', z')$
27                     $open$.DecreasePriority($(l_{t+1},t+1), f'$)
28     return $paths$
```

Algorithm 2: The A* algorithm for the lower level of the pricing problem.

Line 12 calls the A* algorithm to find paths that extend the partial path $((l_0,0),\ldots,(l_t,t))$ to the location L_j of the next vertex j. The pseudocode for this A* algorithm is shown in Algorithm 2. Line 2 creates the output set of paths. Lines 3 and 4 initialize the upper bound on time $\bar{t}$ and dual penalties $\bar{z}$. These values are used to determine if a path lies on the Pareto frontier and will incrementally decrease as the search progresses. Line 5 creates a map data structure that associates every vertex with a label representing a partial path, the accumulation of reduced costs, a lower bound on the reduced cost from start to end and the accumulation of dual penalties. Line 6 creates the root label corresponding to the partial path $((l_0,0),\ldots,(l_t,t))$ input from the high level search. It is initialized with the input reduced cost $\bar{c}$, a lower bound on the total reduced cost estimated using the heuristic h described in Section 4.3.3, and 0 dual penalties so far on this segment. Line 7 creates a priority queue of vertices for future processing. This priority queue prefers vertices whose partial path has a lower reduced-cost-to-go. Line 8 adds the current vertex into the priority queue. Lines 9 to 11 loop over every label. Line 12 proceeds if the partial path is improving. As the search progresses, the time and dual penalties will decrease and the reduced cost will increase. If the partial path reaches the sink after the earliest possible time (Line 13), Lines 14 and 15 update the upper bound on time and dual penalties, and Line 16 stores the partial path.

Line 17 iterates over the outgoing edges. Lines 18 to 20 respectively compute the reduced cost g' of the new partial path from $L_\top$ to l_{t+1}, the reduced-cost-to-go h' from l_{t+1} to L_j and the reduced cost lower bound f' on the complete path from $L_\top$ to L_j. Line 21 calculates the dual penalties incurred on the new path segment. If the new partial path reaches an unseen vertex (Line 22), Lines 23 and 24 store the new path and add it to the priority queue. If the new partial path has a lower reduced-cost-to-go than the existing path ending at the same vertex (Line 25), Lines 26 and 27 store the new path and update the priority in the priority queue. Line 28 returns the set of Pareto-optimal paths.

4.4 Enforcing Integrality

Because the master problem can select paths with fractional proportion, it is embedded in a branch-and-bound tree search to remove these fractionalities. Nodes in the search tree correspond to guesses as to whether an edge in the sequencing graph or the navigation graph is used or not used. Branching rules are subroutines that make these choices.

After the pricers report that sequence-path pairs with negative reduced cost do not exist and the separators report that the master problem solution has no conflicts, then the master problem has been solved at the current node and the branching rules are executed to find and remove a fractionality in the solution by creating children nodes. The following three branching rules are used.

4.4.1 Branching on Edges in the Sequencing Graph

The first branching rule removes fractionalities in the sequences. It selects an edge $e \in \mathcal{E}^{\text{seq}}$ from the sequencing graph that is fractionally used an agent $a \in \mathcal{A}$, i.e.,

$$0 < \sum_{(s,p) \in \Lambda_a} \alpha_s^e \lambda_{a,s,p} < 1$$

where $\alpha_s^e \in \mathbb{Z}_+$ counts the number of times that edge e appears in sequence s. It then creates two children nodes with the constraints

$$\sum_{(s,p) \in \Lambda_a} \alpha_s^e \lambda_{a,s,p} \leq 0 \tag{6a}$$

and

$$\sum_{(s,p) \in \Lambda_a} \alpha_s^e \lambda_{a,s,p} \geq 1. \tag{6b}$$

Constraint (6a) is added to the first child to forbid the agent from taking the edge. Constraint (6b) is added to the second child to force the agent to take the edge. Adding one of these constraints to the master problem also requires its dual variable to be subtracted from the reduced cost function in the pricing problem. This branching rule is called until all edges in the sequencing graph have integer value.

4.4.2 Branching on Path Lengths

The second branching rule branches on the path length of an agent. This branching rule selects a path of length k that is fractionally used by an agent a. In the first child, agent a can only use paths with length k or shorter. In the second child, agent a can only use paths with length greater than k. This is enforced by removing all incompatible paths from the master problem and tightening the time window of the sink $\perp$ when pricing agent a. This branching rule is called until all paths of every agent have identical arrival time at its end location.

4.4.3 Branching on Vertices in the Navigation Graph

The third branching rule is then executed to remove fractionalities in the paths. This branching rule selects a vertex $v^{\text{nav}} \in \mathcal{V}^{\text{nav}}$ of the navigation graph that is fractionally used by an agent a and selects an edge $e^{\text{seq}} = (i, j) \in \mathcal{E}^{\text{seq}}$ of the sequencing graph where v^{nav} appears. The branching rule then creates three children nodes. The first child requires the agent to traverse e^{seq} and visit v^{nav} between i and j. This is enforced by adding one of Constraint (6b) if it has not yet been branched on and forcing the low-level navigation path to visit v^{nav} when extending a partial path from i to j. The second child also requires the agent to traverse e^{seq} but stops it from visiting v^{nav} when extending a partial sequence from i to j. The third child prevents the agent from traversing e^{seq} using Constraint (6a). If decisions about the sequencing edge are incompatible with the earlier decisions, then the new node is infeasible and is not considered further. The three children nodes partition the search space so that any feasible solution will appear in exactly one subtree.

While it is possible to branch solely on the navigation vertex v^{nav} without stipulating during which sequencing edge e^{seq} the visit to v^{nav} occurs, this restriction is not easy to enforce in the pricing problem because the low-level search must consider paths that visit and do not visit v^{nav} in every extension of a partial sequence. As the number of these branching constraints increase deep into the tree, the pricing problem must choose whether to visit or avoid each v^{nav} when finding a negative reduced cost column. Early work indicate that the heuristic of the A* algorithm is extremely difficult to define correctly, leading to a very weak usable heuristic and an excessively slow search in experiments. The three-way branching scheme avoids this combinatorial explosion in the pricing problem and only impacts the combinatorial exploration intrinsic to the branch-and-bound tree of the master problem.

5 BCPB-MAPD

This section presents BCPB-MAPD, the second branch-and-cut-and-price algorithm for optimal MAPD. BCPB-MAPD essentially applies a Dantzig-Wolfe decomposition to solve the PDPTW and then performs a (discrete) Benders decomposition to ensure that solutions to the PDPTW provide an optimal MAPF solution.

The (restricted) master problem of BCPB-MAPD is a similar linear program to the master problem of BCP-MAPD but its variables represent the selection of request sequences and ignores paths. The pricing problem is the resource-constrained shortest path problem common to VRPs and is solved using a standard labeling algorithm. Whenever an integer feasible solution to the master problem is found, the selected request sequences are passed to the Benders problem to find a conflict-free path for each agent that visits the locations of its assigned requests. If feasible collision-free paths do not exist or if the optimal collision-free paths cost more than the PDPTW estimate, then a Benders feasibility cut or optimality cut is added to the master problem, forcing it to choose another set of request sequences.

5.1 The Master Problem

The master problem is similar to BCP-MAPD but its columns represent pure request sequences. Define Λ_a as the set of request sequences for agent $a \in \mathcal{A}$. Associate every sequence $s \in \Lambda_a$ with a cost $c_s \in \mathbb{Z}_+$. Let $\lambda_{a,s} \in [0,1]$ be a decision variable representing the proportion of selecting $s \in \Lambda_a$. The variable $\theta \in \mathbb{R}_+$ represents the additional cost due to collisions in the path finding. The master problem begins as the linear program:

$$\min \sum_{a \in \mathcal{A}} \sum_{s \in \Lambda_a} c_s \lambda_{a,s} + \theta \tag{7a}$$

subject to

$$\sum_{s \in \Lambda_a} \lambda_{a,s} = 1 \qquad \forall a \in \mathcal{A}, \tag{7b}$$

$$\sum_{a \in \mathcal{A}} \sum_{s \in \Lambda_a} \alpha_s^r \lambda_{a,s} = 1 \qquad \forall r \in \mathcal{R}^{\uparrow}, \tag{7c}$$

$$\lambda_{a,s} \geq 0 \qquad \forall a \in \mathcal{A}, s \in \Lambda_a, \tag{7d}$$

$$\theta \geq 0. \tag{7e}$$

Objective Function (7a) minimizes the cost estimated by the PDPTW plus additional costs due to collisions. Constraints (7b) and (7c) are analogous to Constraints (1b) and (1c). Constraints (7d) and (7e) define the variable domains. The master problem can be recognized as the set partitioning master problem commonly seen in VRPs plus the additional θ term.

5.2 Generating Sequences

The sets Λ_a are dynamically built for every agent $a \in \mathcal{A}$ in turn by solving a resource-constrained shortest path problem on $\mathcal{G}^{\text{seq}}$. This problem is essentially identical to the pricing problem commonly seen in VRPs (e.g., Costa, Contardo, and Desaulniers 2019).

First, the pricer pre-computes a lower bound $d_{i,j} = h(L_i, L_j)$ on the distance between any two vertices $i, j \in \mathcal{V}^{\text{seq}}$ by calling an A* algorithm to determine the cost of the shortest path between L_i and L_j, as in Section 4.3.3. Next, a standard labeling algorithm is executed to find a sequence with negative reduced cost. A partial sequence starting and ending at $\top$ is extended outward towards $\perp$. When extending a partial sequence ending at vertex i with reduced cost $\bar{c}_i$ and arrival time τ_i along an edge $(i,j) \in \mathcal{E}^{\text{seq}}$, a new partial sequence ending at j is created with arrival time $\tau_j = \max(\tau_i + d_{i,j}, \underline{T}_j)$ equal to the later of either the earliest arrival time or the time window opening, and with reduced cost $\bar{c}_j = \bar{c}_i + (\tau_j - \tau_i) - \rho_j$ equal to the current reduced cost $\bar{c}_i$ plus the travel and waiting time $\tau_j - \tau_i$ and minus the dual solution ρ_j of Constraint (7c). If the arrival is later than the time window closing (i.e., $\tau_j > \bar{T}_j$), then the partial sequence is infeasible and can be discarded. The standard dominance rules are applicable for this problem. If a sequence ending at $\perp$ has negative reduced cost, it is added to Λ_a together with a new variable.

5.3 Resolving Conflicts

Whenever an integer feasible solution to the master problem is found, a MAPF problem is solved to check if the request sequence assigned to each agent corresponds to a conflict-free path.

Recall that $\alpha_s^e \in \mathbb{Z}_+$ counts the number of times that edge $e \in \mathcal{E}^{\text{seq}}$ appears in sequence s. Let

$$w_a^e = \sum_{s \in \Lambda_a} \alpha_s^e \lambda_{a,s} \tag{8}$$

indicate whether agent $a \in \mathcal{A}$ traverses the edge e. The set

$$\mathcal{W} = \{(a,e) \in \mathcal{A} \times \mathcal{E}^{\text{seq}} : w_a^e = 1\} \tag{9}$$

fully defines the request sequences taken by all agents. The Benders problem runs BCP-MAPD with the sequencing edges fixed according to $\mathcal{W}$, as if by branching. If this MAPF problem is infeasible, a Benders feasibility cut

$$\sum_{(a,e) \in \mathcal{W}} \sum_{s \in \Lambda_a} \alpha_s^e \lambda_{a,s} \leq |\mathcal{W}| - 1$$

is added to the master problem to prohibit this set of edges. If this MAPF problem has a different objective value to the master problem, the objective value of this set of edges is increased by adding a Benders optimality cut

$$\sum_{(a,e) \in \mathcal{W}} \sum_{s \in \Lambda_a} \alpha_s^e \lambda_{a,s} - \frac{1}{\delta}\theta \leq |\mathcal{W}| - 1,$$

where $\delta \in \mathbb{Z}_+$ is the difference between the cost of the PDPTW solution and the MAPF solution. Typical of Benders decomposition, this model views the PDPTW as a relaxation of the MAPD problem.

The Benders subproblem cleanly disconnects the MAPD problem into a PDPTW and a MAPF problem and hence allows for sophisticated PDPTW techniques (e.g., bidirectional search, ng routes, reduced cost fixing, etc.) and MAPF techniques (e.g., goal conflict constraints, reservation table, solution caching, etc.) to be implemented more easily. If implemented in BCP-MAPD, these techniques would interact with too many moving parts and make for a very difficult and hence bug-prone implementation. To maintain a simple code and to have a fair comparison between BCP-MAPD and BCPB-MAPD, these techniques are not implemented for the experiments.

5.4 Branching Rules

In the master problem, the branching rule for sequencing edges (Section 4.4.1) is used. In the Benders problem, the two MAPF branching rules (Sections 4.4.2 and 4.4.3) are used.

6 Experiments

This section presents the experimental results.

6.1 Set-Up

The instances are generated by extending the standard MAPF benchmarks. MAPF instances only contain agent data, comprising a start and end location. These instances are complemented with randomly generated orders, each consisting of a pickup and delivery location and time window. Given the finite time windows and a finite number of agents, some instances could be infeasible.

Two warehouse maps (Li et al. 2019, 2020), one city map (Stern et al. 2019) and one computer game map (Stern et al. 2019) are chosen. Different numbers of agents and orders (pickup-delivery pairs) are run for each of the four maps. Twenty instances are run for every combination of agents, orders and map. The experiments are conducted over a total of 1,600 instances. Every instance is solved with a time limit of 1 hour on an AMD EPYC Rome CPU at 2.25 GHz with 128 GB of main memory.

BCP-MAPD and BCPB-MAPD are compared against a baseline two-stage heuristic that first solves the PDPTW and then uses the optimal solution to solve a MAPF problem. The implementation essentially modifies BCPB-MAPD to call the MAPF solver only on the optimal VRP solution, instead of on every feasible solution. All three solvers are coded in C++ and use SCIP 8.0.3 for branch-and-bound and Gurobi 10.0.1 for the linear programming restricted master problem.

	BCP-MAPD	BCPB-MAPD	Two-Stage
Optimal	489 (31%)	449 (28%)	114 (7%)
Feasible	571 (36%)	1171 (73%)	1207 (75%)
Found Gap	571 (36%)	1136 (71%)	1207 (75%)
Mean Gap	0.2%	4.0%	4.2%
Time (s)	2715	2759	1443

Table 1: Summary statistics from the experiments.

6.2 Comparing the Algorithms

Table 1 reports overall statistics for the three algorithms. BCP-MAPD, BCPB-MAPD and the two-stage heuristic respectively find provably optimal solutions to 31%, 28% and 7% of the 1,600 instances and feasible solutions to 36%, 73% and 75% of the instances. None of the instances are proven to be infeasible (although none of them are confirmed to be infeasible either). Moving from the two-stage heuristic to BCPB-MAPD to BCP-MAPD can be conceptually thought of as optimizing more and more of the problem simultaneously. As intuition suggests, the results confirm that more joint optimization is beneficial at finding optimal solutions, while less joint optimization is better at finding feasible solutions due to the difficulty of simultaneously optimizing two NP-hard problems.

A lower bound and upper bound, and hence an optimality gap, are found for 36%, 71% and 75% of the instances by BCP-MAPD, BCPB-MAPD and the two-stage heuristic. There are 511 instances for which all algorithms found an optimality gap, and on these instances, the average gap is 0.2%, 4.0% and 4.2% respectively. The average run time is 2715, 2759 and 1443 seconds, where timed out instances are counted as 3600 seconds.

The results indicate that jointly optimizing the sequencing and navigation achieves the best average results. However, these findings are highly dependent on the map structure. Figure 3 shows the percentage of instances solved optimally by the three solvers on each map.

The first map, 10x30-w5, is a small map that contains narrow corridors that replicate warehouses. Given the large percentage of locations being obstacles, the agents have very little space to move around and hence face high congestion. Therefore, the PDPTW relaxation gives a poor lower bound and consequently the two-stage heuristic fails to prove optimality or infeasibility on any instance. BCPB-MAPD, which is based on similar two-stage ideas, also declines very quickly as the number of agents increase. In contrast, BCP-MAPD scales significantly better and dominates the other two algorithms.

The second map, 31x79-w5, is a medium-size map that is also modeled on warehouses. The two-stage heuristic could prove optimality on three of the twenty instances with 20 agents and 10 orders, and on one of the instances with 20 agents and 50 orders. Similar to the smaller warehouse, BCPB-MAPD fails at scaling beyond the small instances and is dominated by BCP-MAPD.

The third map, Berlin_1_256, is a medium-size map with immense open spaces for agents to wander but it contains several choke points. Given the availability of space, the agents rarely encounter each other and hence the VRP relaxation gives an optimal lower bound for some of the smaller instances. The two-stage heuristic proves optimality for a surprising number of instances on this map and even solves a few more instances than BCP-MAPD at higher order counts. The performance characteristics of BCP-MAPD and BCPB-MAPD are reversed compared to the warehouse maps. BCPB-MAPD holds steady up to 60 agents and only then collapses. The time spent in the navigation graph by BCP-MAPD is clearly not productive when the map is very sparse because the VRP solutions provide a strong lower bound.

The fourth map, den312d, is a small map with large open spaces but does not contain narrow choke points. The two optimal algorithms perform similarly but BCPB-MAPD has a slight advantage, presumably due to the large open spaces again. The two-stage heuristic is almost competitive at 10 agents but deteriorates from 15 agents.

Overall, these experiments indicate that the integrated algorithm BCP-MAPD solves more instances when agents are gridlocked because combining the task assignment and path finding is critical in obtaining a tight lower bound and feasible solutions. In contrast, BCPB-MAPD solves more instances with less crowding because the VRP relaxation is tight and hence it requires very few calls to the path finding solver. The two-stage heuristic performs remarkably well for the same reason. It also completes in significantly less time.

6.3 Allowing Vacant Agents to Move Under Shelves

In automated warehouses, the robots lift up the shelves from underneath. While a robot is not carrying a shelf, it can travel under shelves, opening up large swathes of space for navigation. In a more general problem to be considered in future work, robots should be allowed to move shelves around the map to open up new corridors. This fully cooperative problem will be exceedingly difficult given the synchronization of robots and movable shelves. This section considers a simple extension of MAPD that partially mimics the more general problem described above by allowing agents to move under obstacles while they are not carrying an item. In this problem, the obstacles do not move but merely that the agents can move through the obstacles while not carrying an item.

Figure 4 shows the average difference in the optimal objective value on the 284 instances proven optimal by BCP-MAPD on both the original MAPD problem and the modified problem. The number of instances used to compute the average is shown above each point. There are not many instances solved optimally for the larger warehouse 31x79-w5, making the results difficult to interpret. On the smaller warehouse 10x30-w5, allowing agents to move under obstacles does not seem to have significant impact on the total cost. Presumably this is due to both the small number of agents and the Manhattan distance metric, in which the agents can use one of many symmetric paths with identical costs (e.g., an agent moving west then north has the same cost as moving north then west). This kind of symmetry is also exposed in earlier work evaluating heuristics for MAPF (Lam et al. 2023).

7 Conclusion and Future Directions

This paper presents two branch-and-cut-and-price algorithms for MAPD named BCP-MAPD and BCPB-MAPD. BCP-MAPD includes a novel pricing problem that performs a two-level search for a sequence of requests and a path that connects these requests. BCPB-MAPD first solves the sequencing problem and uses combinatorial Benders decomposition to defer the path finding problem until a feasible sequencing solution is found.

Experimental results indicate that neither algorithm dominates but BCP-MAPD has an advantage in crowded environments whereas BCPB-MAPD proved beneficial in less crowded environments because the agents rarely encounter each other and hence the optimal VRP solution provides a tight lower bound for MAPF. Because of this property, the two-stage sequence-first and navigate-second heuristic performs unexpectedly well, obtaining a provably optimal solution for several small instances.

Experiments are also carried out to evaluate the impact of allowing agents to move under obstacles while not carrying an item. This modification models robots moving under shelves in warehouses. Empirical results demonstrate that the impact is insignificant, presumably due to the small instance sizes and the symmetries present in the Manhattan distance metric.

Recent algorithms for MAPF are starting to tackle industrial-size instances with a few hundred agents, which were intractable just a few years ago. The immediate research direction for MAPD is to similarly scale up to a few hundred agents and orders. Seven decades of work on VRPs are now producing exact algorithms that can optimize over a thousand customers, suggesting that equivalent progress for MAPD is difficult but achievable. Such an outcome would make exact techniques much more relevant to industry deployment in automated warehousing.

Acknowledgement

This research is supported by the Australian Research Council under Discovery Grants DP190100013 and DP200100025 and by a gift from Amazon.

References

Baldacci R, Bartolini E, Mingozzi A, 2011 *An exact algorithm for the pickup and delivery problem with time windows. Operations Research* 59(2):414–426.

Berbeglia G, Cordeau JF, Gribkovskaia I, Laporte G, 2007 *Static pickup and delivery problems: a classification scheme and survey. TOP* 15(1):1–31.

Chen Z, Alonso-Mora J, Bai X, Harabor DD, Stuckey PJ, 2021 *Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. IEEE Robotics and Automation Letters* 6(3):5816–5823.

Codato G, Fischetti M, 2006 *Combinatorial Benders' cuts for mixed-integer linear programming. Operations Research* 54(4):756–766.

Corréa AI, Langevin A, Rousseau LM, 2007 *Scheduling and routing of automated guided vehicles: A hybrid approach. Computers & Operations Research* 34(6):1688–1707.

Costa L, Contardo C, Desaulniers G, 2019 *Exact branch-price-and-cut algorithms for vehicle routing. Transportation Science* 53(4):946–985.

Davies TO, Gange G, Stuckey PJ, 2017 *Automatic logic-based Benders decomposition with MiniZinc. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 787–793.

Desaulniers G, Langevin A, Riopel D, Villeneuve B, 2003 *Dispatching and conflict-free routing of automated guided vehicles: An exact approach. International Journal of Flexible Manufacturing Systems* 15(4):309–331.

Desrosiers J, Lübbecke ME, 2010 *Branch-price-and-cut algorithms. Wiley Encyclopedia of Operations Research and Management Science* (John Wiley & Sons, Inc.).

Dumas Y, Desrosiers J, Soumis F, 1991 *The pickup and delivery problem with time windows. European Journal of Operational Research* 54(1):7–22.

Feillet D, Dejax P, Gendreau M, Gueguen C, 2004 *An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. Networks* 44(3):216–229.

Fukasawa R, Longo H, Lysgaard J, De Aragão MP, Reis M, Uchoa E, Werneck RF, 2006 *Robust branch-and-cut-and-price for the capacitated vehicle routing problem. Mathematical Programming* 106(3):491–511.

Gange G, Harabor D, Stuckey P, 2019 *Lazy CBS: Implicit conflict-based search using lazy clause generation. Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, volume 29, 155–162.

Helsgaun K, 2017 *An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems.* Technical report, Roskilde University.

Henkel C, Abbenseth J, Toussaint M, 2019 *An optimal algorithm to solve the combined task allocation and path finding problem. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 4140–4146.

Hooker JN, Ottosson G, 2003 *Logic-based Benders decomposition. Mathematical Programming* 96(1):33–60.

Jepsen M, Petersen B, Spoorendonk S, Pisinger D, 2008 *Subset-row inequalities applied to the vehicle-routing problem with time windows. Operations Research* 56(2):497–511.

Lam E, Gange G, Stuckey PJ, Van Hentenryck P, Dekker JJ, 2020 *Nutmeg: A MIP and CP hybrid solver using branch-and-check. SN Operations Research Forum* 1(3):22.

Lam E, Harabor D, Stuckey PJ, Li J, 2023 *Exact anytime multi-agent path finding using branch-and-cut-and-price and large neighborhood search. Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS).*

Lam E, Le Bodic P, 2020 *New valid inequalities in branch-and-cut-and-price for multi-agent path finding. Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, 184–192.

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

Lam E, Van Hentenryck P, 2017 *Branch-and-check with explanations for the vehicle routing problem with time windows. Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, 579–595 (Springer).

Letchford AN, Salazar-González JJ, 2006 *Projection results for vehicle routing. Mathematical Programming* 105(2):251–274.

Li J, Gange G, Harabor D, Stuckey PJ, Ma H, Koenig S, 2020 *New techniques for pairwise symmetry breaking in multi-agent path finding. Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 193–201.

Li J, Harabor D, Stuckey P, Felner A, Ma H, Koenig S, 2019 *Disjoint splitting for multi-agent path finding with conflict-based search. Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS).*

Li J, Harabor D, Stuckey PJ, Ma H, Gange G, Koenig S, 2021 *Pairwise symmetry reasoning for multi-agent path finding search. Artificial Intelligence* 301:103574.

Liu M, Ma H, Li J, Koenig S, 2019 *Task and path planning for multi-agent pickup and delivery. Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 1152–1160.

Lübbecke ME, Desrosiers J, 2005 *Selected topics in column generation. Operations Research* 53(6).

Ma H, Li J, Kumar TKS, Koenig S, 2017 *Lifelong multi-agent path finding for online pickup and delivery tasks. Proceedings of the International Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 837–845.

Ren Z, Rathinam S, Choset H, 2023 *CBSS: A new approach for multiagent combinatorial path finding. IEEE Transactions on Robotics* 1–15.

Røpke S, Cordeau JF, 2009 *Branch and cut and price for the pickup and delivery problem with time windows. Transportation Science* 43(3):267–286.

Sharon G, Stern R, Felner A, Sturtevant N, 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, Boyarski E, Bartak R, 2019 *Multi-agent pathfinding: Definitions, variants, and benchmarks. Proceedings of the Symposium on Combinatorial Search (SoCS)*, 151–158.

Vigo D, Toth P, eds., 2014 *Vehicle Routing: Problems, Methods, and Applications.* MOS-SIAM Series on Optimization (Society for Industrial and Applied Mathematics), 2nd edition.

Wolsey LA, 2021 *Integer Programming* (Wiley), 2nd edition.

Xu Q, Li J, Koenig S, Ma H, 2022 *Multi-goal multi-agent pickup and delivery. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9964–9971.

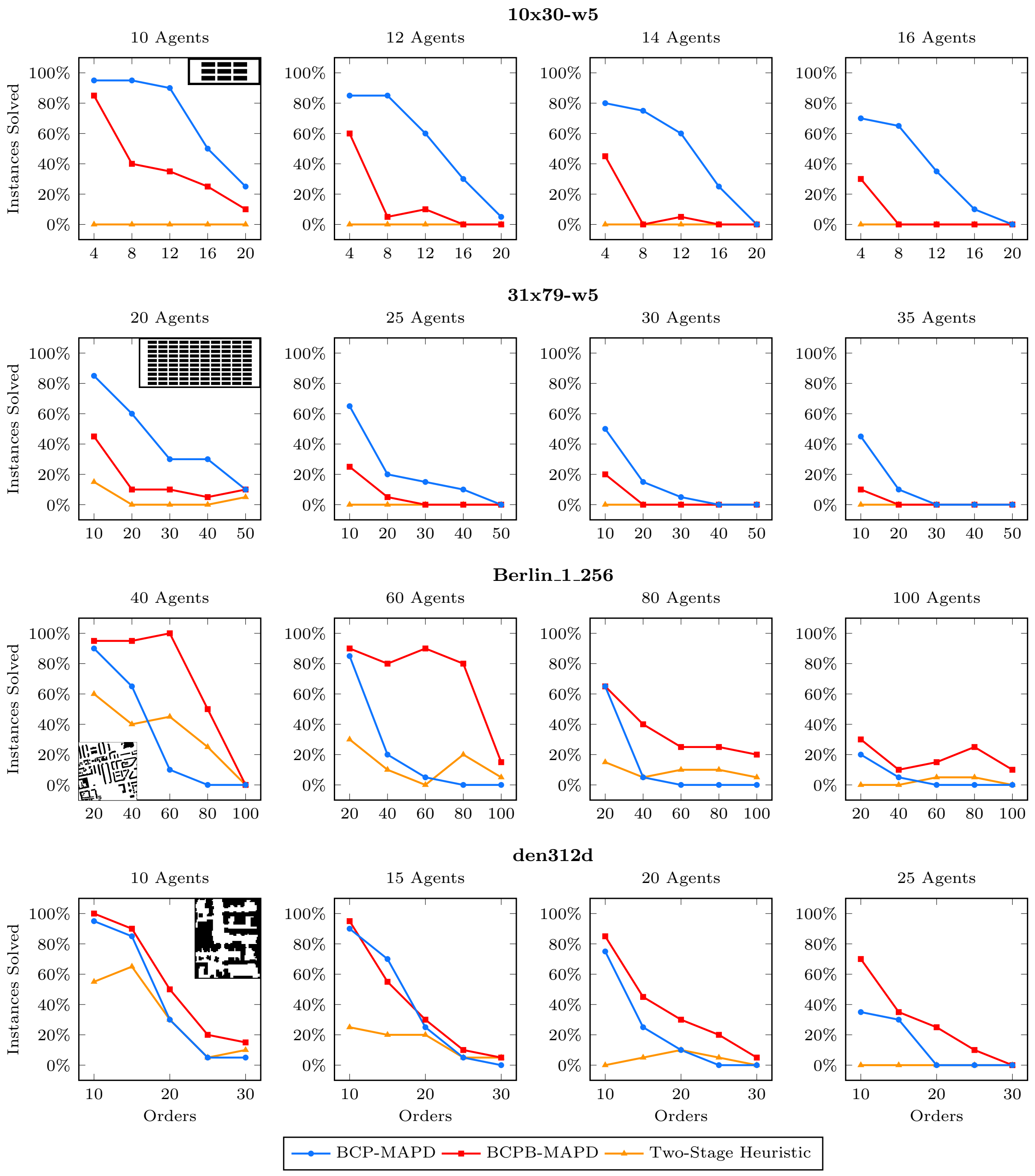

Figure 3: Percentage of instances solved, plotted for different maps, number of agents and number of orders (pickup-delivery pairs). Higher is better.

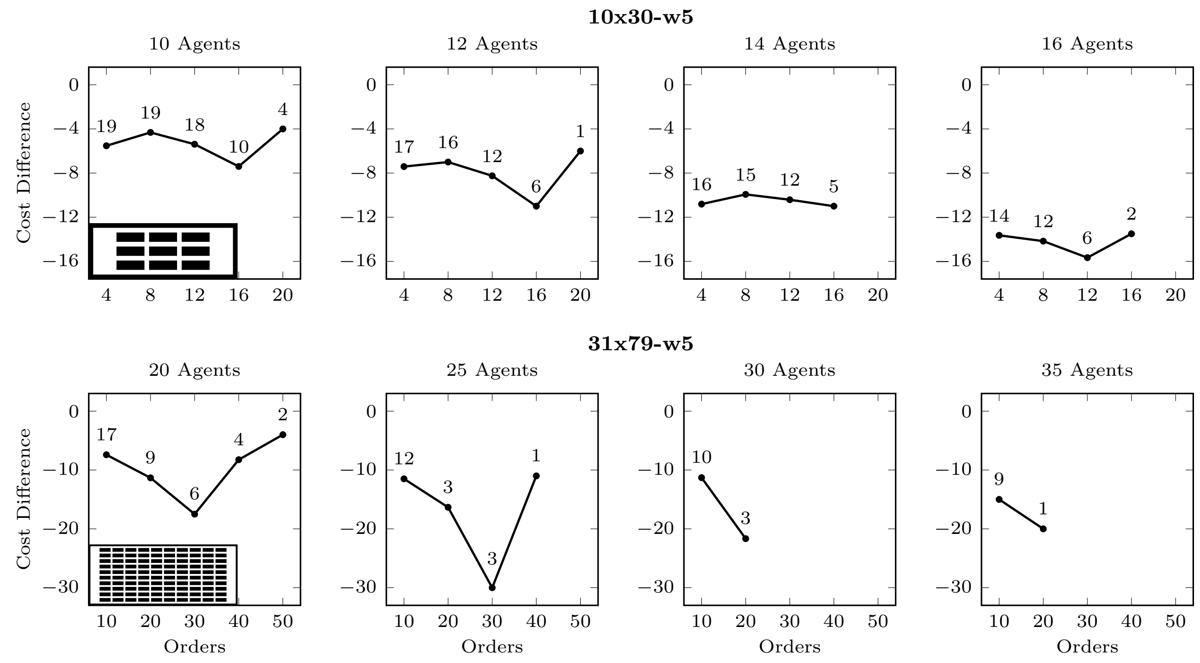

Figure 4: Average difference in cost after allowing agents to move through obstacles while not carrying an item. Every point is labeled with the number of instances averaged over.