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

Edward Lam, Peter J. Stuckey, and Daniel Harabor

Monash University

September 12, 2024

Abstract

Given a set of agents and a set of pickup-delivery requests located on a two-dimensional grid 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 other agents 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 solved simultaneously 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 exist. 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, significantly higher than a state-of-the-art suboptimal approach.

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 agent is 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 dense warehouse map. The deferred path finding algorithm achieves an average optimality gap of 4.2%, solves more instances with lower congestion and could reach 100 agents and 100 orders on a sparse computer game map.

The contributions of this paper are (1) two algorithms believed to be the first that solve MAPD optimally, (2) an intricate pricing algorithm for optimizing the two-level simultaneous sequencing and path finding problem and (3) computational results showing that the two proposed algorithms scale substantially better than CBSS, a state-of-the-art suboptimal method for a closely related problem.

The remainder of the paper is organized as follows. Section 2 reviews relevant techniques and related work. Section 3 formalizes the problem. Sections 4 and 5 describe BCP-MAPD and BCPB-MAPD. Section 6 presents the experimental results. Section 7 concludes this paper and discusses directions for future research.

2 Background and Related Work

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 imposed in both the master problem and the pricing problem. In the second variant, the elementarity constraint is only imposed 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. In the basic MAPF problem, agents are assumed to travel at a constant speed. There are extensions of MAPF that model the physical aspects of robot motion, such as the geometry of large robots (Li et al. 2019b) and the translational and rotational speed (Hoenig et al. 2016). See the survey by Stern et al. (2019) for a detailed taxonomy of MAPF variants.

The current state-of-the-art for exact MAPF consists of 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 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. A combination of Dantzig-Wolfe and Benders decomposition is the foundation of one algorithm presented in this paper. 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 study variations of MAPD under different names and hence make qualitative comparisons difficult and quantitative comparisons impossible. 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. This work appears to tackle the problem most similar to MAPD. They introduce CBSS, a tree search algorithm based on CBS, that calls an external Traveling Salesman solver. While CBSS is theoretically optimal, the implementation unfortunately executes the Lin-Kernighan-Helsgaun heuristic (Helsgaun 2017) for solving the Traveling Salesman Problems and hence voids all optimality guarantees. Their experiments show that CBSS finds feasible solutions to instances with up to 20 agents and 50 requests, while a baseline optimal A* algorithm in the joint space is unable to solve any instance with 5 agents.

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 cutting planes, branching rules, etc. We also exploit the grid layout inherited from MAPF to implement additional symmetry-breaking techniques (e.g., rectangle cuts).

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 integer coordinates. Non-traversable locations, called obstacles, are omitted from $\mathcal{L}$. For any location $l_i = (x_i, y_i) \in \mathcal{L}$, define $\delta(l_i) = \{(x_j, y_j) \in \mathcal{L} : |x_i - x_j| + |y_i - y_j| \leq 1\}$ as the neighbors of l_i. That is, the neighbors of a location are itself and the four locations north, south, east and west of the location.

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 $\mathcal{E} = \{((l_i, t), (l_j, t+1)) \in \mathcal{V} \times \mathcal{V} : l_j \in \delta(l_i)\}$ 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 $e' = ((l_j, t), (l_i, t+1))$ of an edge $e = ((l_i, t), (l_j, t+1))$ 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 $L_a^- \in \mathcal{L}$. All start locations are unique and all end locations are unique, but it is possible for an agent to have its start location be its end location, and the start (resp. end) location of an agent be the end (resp. start) location of another agent.

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 $\mathcal{O} = \{(r_1^+, r_1^-), \ldots, (r_O^+, r_O^-)\}$ be the set of orders, where an order is a pair of a pickup request and a delivery request. Define $\mathcal{R}^+ = \{r_1^+, \ldots, r_O^+\}$ and $\mathcal{R}^- = \{r_1^-, \ldots, r_O^-\}$ as the set of pickup requests and delivery requests respectively. Every request $r \in \mathcal{R}^+ \cup \mathcal{R}^-$ 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 $(r^+, r^-) \in \mathcal{O}$ is assigned to an agent, then the path assigned to the agent must visit the vertices $(L_{r^+}, t_{r^+}) \in \mathcal{V}$ and $(L_{r^-}, t_{r^-}) \in \mathcal{V}$ at some time $t_{r^+} \in \{\underline{T}_{r^+}, \ldots, \bar{T}_{r^+}\}$ and $t_{r^-} \in \{\underline{T}_{r^-}, \ldots, \bar{T}_{r^-}\}$, where $t_{r^+} \leq t_{r^-}$. At any given time, an agent can undertake at most one order, i.e., $t_{r_i^-} \leq t_{r_j^+}$ or $t_{r_j^-} \leq t_{r_i^+}$ for any two distinct orders $(r_i^+, r_i^-), (r_j^+, r_j^-) \in \mathcal{O}$ 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 $p_i = (l_0^{p_i}, \ldots, l_{k_1-1}^{p_i})$ and a different agent takes the path $p_j = (l_0^{p_j}, \ldots, l_{k_2-1}^{p_j})$, the conditions $l_t^{p_i} \neq l_t^{p_j}$ and $l_t^{p_i} \neq l_{t+1}^{p_j} \vee l_t^{p_j} \neq l_{t+1}^{p_i}$ 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 $\mathcal{V}^{\text{seq}} = \mathcal{R}^+ \cup \mathcal{R}^- \cup \{\top, \bot\}$ where $\top$ and $\bot$ are source and sink vertices representing the start and end location of an agent. The edges

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

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 $s = (\top, r_1, \ldots, r_n, \bot)$ as a path on $\mathcal{G}^{\text{seq}}$ from the source $\top$ to the sink $\bot$, where $r_1, \ldots, r_n \in \mathcal{R}^+ \cup \mathcal{R}^-$ are the pickup and delivery requests completed in s. The requests $r_1, \ldots, r_n$ are not necessarily unique (i.e., a request can be completed more than once within a sequence) and rely on the master problem to select sequences in which every request is completed exactly once.

Every request sequence s is associated with an agent $a \in \mathcal{A}$ and a path $p = (l_0, \ldots, l_{k-1})$ of length k on $\mathcal{G}^{\text{nav}}$ that navigates the agent to the locations of the requests $r_1, \ldots, r_n$, 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 $i \in \{1 \ldots, n\}$. 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 $\Lambda_a = \{(s_1^a, p_1^a), \ldots, (s_{|\Lambda_a|}^a, p_{|\Lambda_a|}^a)\}$ 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 $r \in \mathcal{R}^+ \cup \mathcal{R}^-$

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

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

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

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

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

At this stage, the master problem does not impose constraints on the locations visited by the agents and therefore feasible solutions can contain vertex conflicts and edge conflicts. The following constraints prevent several types of conflicts.

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). 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. An edge conflict at e can be prevented using the constraint

$$\sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} (\alpha_p^e + \alpha_p^{e'}) \lambda_{a,s,p} \leq 1. \tag{2}$$

Because the time expansion produces a large number of edges but very few of these edges participate in an edge conflict, none of Constraint (2) are added upfront but rather they are generated on demand by a subroutine called a separator.

Whenever a feasible solution to the master problem is found, the edge conflict separator checks it for edge conflicts and resolves these conflicts by adding constraints to the master problem, forcing it to choose different sequence-path pairs. The separator first computes the number x_e of times that an edge e or its reverse e' are traversed by all agents:

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

Whenever $x_e > 1$, an edge conflict occurs at e and Constraint (2) is added to the master problem to resolve this edge conflict.

4.2.2 Vertex Conflicts

A vertex conflict occurs when two or more agents enter the same (time-expanded) vertex. Equivalently, a vertex conflict occurs whenever the five incoming edges to a vertex (i.e., the edges originating in the north, south, west and east directions and the wait action at the previous timestep) are used more than once. For any vertex $v = (l,t) \in \mathcal{V}^{\text{nav}}$, define

$$\alpha_p^v = \sum_{l_i \in \delta(l)} \alpha_p^{((l_i,t-1),(l,t))}$$

as the number of times that path p visits vertex v. Vertex conflict constraints are again generated dynamically by a separator due to the time expansion. The separator first calculates the number x_v of times that vertex v is visited by all agents:

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

Whenever $x_v > 1$, a vertex conflict occurs at v and the constraint

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

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

4.2.3 General Form of Conflict Constraints

Vertex, edge and many other conflict constraints can be expressed in a common form. Let $\mathcal{B}$ be the set of conflict constraints. Every conflict constraint $b \in \mathcal{B}$ can be defined by 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

$$\sum_{a \in \mathcal{A}} \sum_{(s,p) \in \Lambda_a} \left(\sum_{e \in \mathcal{E}^{\text{nav}}} \beta_b^{a,e} \alpha_p^e \right) \lambda_{a,s,p} \leq \beta_b. \tag{4}$$

Constraints in this form are described as *robust* (de Aragao and Uchoa 2003, Fukasawa et al. 2006). Whenever a robust cut is added to the master problem, the pricing problem is simply modified with one additional term in the objective function. In contrast, adding a non-robust cut could potentially require a completely different pricing problem.

4.2.4 Other Cuts

Including constraints for removing vertex conflicts and edge conflicts is sufficient for correctly solving the MAPD problem. However, as is standard practice in cutting planes methods for integer programming, the master problem can be tightened using other constraints that provide additional reasoning.

Lam and Le Bodic (2020) and Lam et al. (2022) developed eleven families of valid inequalities for MAPF but showed that only some significantly improved solve time. BCP-MAPD reimplements the MAPF rectangle, exit entry, wait corridor, wait edge and wait two-edge cuts, all of which are robust and can be expressed in the form of Constraint (4). The wait edge constraints are a lifting of the edge constraints and therefore using them negates the need for Constraint (2). Note that these cuts rely on the two-dimensional rectangular-shaped map, as defined in Section 3. The non-robust goal cuts also have a significant impact on MAPF solve times but handling their dual variables in the two-level pricing problem of MAPD proved too difficult and therefore these cuts 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 (e.g., multi-star, hypotour, etc.) 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 common in VRPs, as well as a *contention penalties* resource that corresponds to the sum of negative dual values incurred along a partial path. The contention penalties represent the extra cost for using a location in contention with another agent.

The high-level problem is solved using 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 $t_j \in \{\underline{T}_j, \ldots, \bar{T}_j\}$, where the source and sink locations $L_\top := L_a^+$ and $L_\bot := L_a^-$ are the start and end locations of agent a. This path segment navigates the agent from its current location to the location of the next request or its end location.

The low-level problem maintains resources but does not have resource constraints because time is encoded in the time-expanded graph $\mathcal{G}^{\text{nav}}$ and the other resource constraints are considered in the high-level problem. When solving the low-level shortest path problem, every Pareto-optimal path corresponds to a new partial sequence in the high-level problem. A Pareto frontier of low-level paths is necessary to guarantee optimality of the pricing problem because reduced cost, time and contention penalties 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 and Contention Penalties

While any sequence-path pair with negative reduced cost can be added as 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 is that finding a column with the most negative reduced cost minimizes the master problem as quickly as possible, mirroring the pivot rule in a rudimentary simplex implementation.

Let $\pi_a, \rho_r \in \mathbb{R}, \sigma_b \in \mathbb{R}_-$ be the dual variables of Constraints (1b), (1c) and (4) respectively. When solving the pricing problem for agent $a \in \mathcal{A}$, a sequence-path pair (s, p) comprising sequence $s = (\top, r_1, \ldots, r_n, \bot)$ and path $p = (l_0, \ldots, l_{k-1})$ has reduced cost

$$\bar{c}_{(s,p)} = c_p - \pi_a - \sum_{i=1}^{n} \rho_{r_i} - \sum_{t=0}^{T-2} \sum_{b \in \mathcal{B}} \beta_b^{a,((l_t,t),(l_{t+1},t+1))} \sigma_b. \tag{5}$$

Define $z_p = -\sum_{t=0}^{T-2} \sum_{b \in \mathcal{B}} \beta_b^{a,((l_t,t),(l_{t+1},t+1))} \sigma_b$ as the contention penalties of path p, and $\bar{c}_s = -\pi_a - \sum_{i=1}^{n} \rho_{r_i}$ and $\bar{c}_p = c_p + z_p$ respectively as the contributions of sequence s and path p to the reduced cost of sequence-path pair (s, p).

Notice that the dual values $\sigma_b \leq 0$ of Constraint (4) are always non-positive and therefore the contention penalties $z_p = -\sum_{t=0}^{T-2} \sum_{b \in \mathcal{B}} \beta_b^{a,((l_t,t),(l_{t+1},t+1))} \sigma_b \geq 0$. Hence, $\bar{c}_p = c_p + z_p \geq c_p$ and the true cost c_p is a lower bound on the reduced cost $\bar{c}_p$.

4.3.2 Encoding Reduced Costs on the Networks

The accumulation of reduced costs 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 $r \in \mathcal{R}^+$ 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 $-\sum_{b \in \mathcal{B}} \beta_b^{a,e} \sigma_b$ corresponding to the contention penalties.

Recall from Section 3 that an agent a taking a path $p = (l_0, \ldots, l_{k-1})$ of length k will traverse the edges $((l_t, t), (l_{t+1}, t+1))$ for all $t \in \{0, \ldots, k-2\}$, where $l_0 = L_a^+$ and $l_{k-1} = L_a^-$, and then the edges $((L_a^-, t), (L_a^-, t+1))$ for all $t \in \{k-1, \ldots, T-2\}$ 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 $t \in \{k-1, \ldots, T-2\}$, 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 $s = (\top, r_1, \ldots, r_n)$ ending at the request r_n is extended to the sink $\bot$, 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

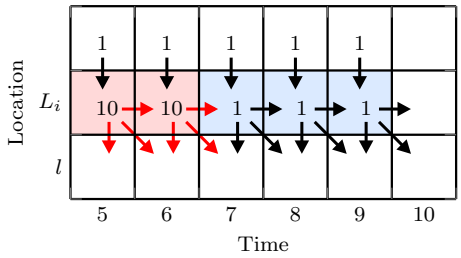

Figure 1: The vertex $i \in \mathcal{V}^{\text{seq}}$ is associated with two time ranges $[5, 6]$ and $[7, 9]$. Within each of these time ranges, the location L_i has identical reduced costs on all outgoing edges on the navigation graph.

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 $r \in \mathcal{R}^+ \cup \mathcal{R}^-$, the original graph $\mathcal{G}^{\text{nav}}$ is used.

4.3.3 Reduced Costs on Outgoing Edges at Request Locations

As the high-level sequencing search finds a sequence from one request to the next request, it directs the low-level navigation search to find a path from the location of the first request to the location of the next request. This myopic low-level search will not see outgoing reduced costs at the destination and fail to account for them in the Pareto frontier. This issue is addressed by augmenting the locations of requests with time intervals. During each of these time intervals, all outgoing edges on the navigation graph must have identical reduced costs. The high-level sequencing search then directs the low-level navigation search to find paths to the request location within every time interval, instead of to the request location at any time during its time window. The purpose of these time intervals is better explained in the following example.

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 $\mathcal{I}_i = \{(\underline{t}_1, \bar{t}_1), \ldots, (\underline{t}_{|\mathcal{I}_i|}, \bar{t}_{|\mathcal{I}_i|})\} \subset \mathcal{T} \times \mathcal{T}$ whose elements $(\underline{t}, \bar{t}) \in \mathcal{I}_i$ represent maximal time ranges such that all outgoing edges from $(L_i, t) \in \mathcal{V}^{\text{nav}}$, $t \in \{\underline{t}, \ldots, \bar{t}\}$, 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 $(L_i, 7)$.

4.3.4 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 $i \in \mathcal{V}^{\text{seq}}$, let $\bar{c}_s$, τ_s and z_s respectively be its reduced cost, arrival time and contention penalties. Consider two partial sequences s_1 and s_2 ending at a common vertex i with resource consumptions $\bar{c}_{s_1}, \tau_{s_1}, z_{s_1}$ and $\bar{c}_{s_2}, \tau_{s_2}, z_{s_2}$. The sequence s_1 *dominates* s_2 and s_2 can be discarded from further consideration if

$$\tau_{s_1} \leq \tau_{s_2}, \tag{6a}$$

$$z_{s_1} \leq z_{s_2}. \tag{6b}$$

and

$$\bar{c}_{s_1} + \sum_{t=\tau_{s_1}}^{\tau_{s_2}-1} \left(1 - \sum_{b \in \mathcal{B}} \beta_b^{a,((L_i,t),(L_i,t+1))} \sigma_b\right) \leq \bar{c}_{s_2}. \tag{6c}$$

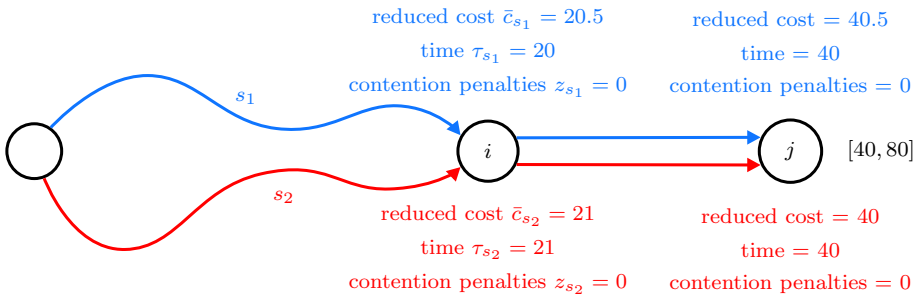

Figure 2: Two sequences demonstrating that the standard dominance rules are not valid for the navigation problem.

Conditions (6a) and (6b) are the usual dominance conditions, which state that s_1 can dominate s_2 if s_1 arrives before or at the same time as s_2, and s_1 has been penalized less than s_2. Condition (6c) 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, for any extension of s_2 to the goal, waiting after s_1 from time τ_{s_1} to τ_{s_2} and then concatenating the extension gives a cheaper sequence than the same extension on s_2.

Figure 2 shows a counterexample of why the sum in Condition (6c) is necessary. Two sequences s_1 and s_2 arrive at vertex i respectively with $\bar{c}_{s_1} = 20.5, \tau_{s_1} = 20, z_{s_1} = 0$ (the 0.5 in $\bar{c}_{s_1} = \tau_{s_1} + 0.5$ could come from cuts or Constraint (1c), not z_{s_1}), and $\bar{c}_{s_2} = 21, \tau_{s_2} = 21, z_{s_2} = 0$. According to the classical dominance rules $\bar{c}_{s_1} \leq \bar{c}_{s_2}$, $\tau_{s_1} \leq \tau_{s_2}$ and $z_{s_1} \leq z_{s_2}$, 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 (6c) 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 constructed.

Condition (6c) can be strengthened by using the reduced cost of the actual shortest path from (L_i, τ_{s_1}) to (L_i, τ_{s_2}) but repeatedly computing the cost of this path is prohibitively expensive because the dominance check is performed for a huge number of partial sequences. Therefore, the weaker Condition (6c) is used instead. For the vast majority of cases, waiting is likely to be optimal anyway.

4.3.5 Preventing Subtours in the Sequencing Level

Labeling algorithms for the pricing problem in VRPs often include one binary resource for every request to indicate whether the request has been completed (e.g., 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 solve time. In contrast, preliminary experiments for MAPD conclusively demonstrate that including request indicator resources to enforce elementarity in pricing substantially worsens performance and hence elementarity is only enforced in the master problem. In the absence of these resources, a sequence can visit a request vertex multiple times but it will be infeasible in an integer solution 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.

4.3.6 Ignoring Reduced Costs in the Navigation Level

The low-level navigation problem searches for path segments between locations as directed by the high-level sequencing problem. Whenever extending a partial sequence from one request vertex to the next, the navigation problem finds a set of Pareto-optimal path segments across the reduced cost, time and contention penalties resources that each navigates the agent from its current location to the location of the next request. Each of these path segments is then concatenated on the partial path associated with the existing sequence, forming a set of new partial sequences that each must be further extended.

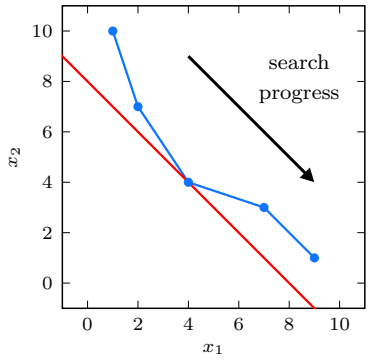

Figure 3: A set of Pareto-optimal points over $x_1, x_2 \geq 0$ (blue). The point $(4, 4)$ is also Pareto-optimal on $x_3 = x_1 + x_2$ (red for $x_3 = 8$).

Recall from Section 3 that the cost c_p of a path p is the time that the agent reaches its end location and waits there indefinitely. Since the path reduced cost $\bar{c}_p = c_p + z_p$ is a monotonic function of the non-negative completion time $c_p \geq 0$ and the contention penalties $z_p \geq 0$, a Pareto frontier of path segments on the two dimensions c_p and z_p will necessarily contain one that minimizes the reduced cost $\bar{c}_p$. Therefore, an algorithm for the navigation problem can ignore the reduced cost resource and search only on the time and contention penalties resources.

Searching for a two-dimensional Pareto frontier, as opposed to a higher-dimensional frontier, is particularly nice because a best-first search can find the frontier simply by focusing on one dimension and progressively enforcing a tighter upper bound on the other dimension. Figure 3 illustrates an example. Consider a two-dimensional Pareto frontier on $x_1, x_2 \geq 0$ and define $x_3 = x_1 + x_2$. When using best-first search to minimize x_1, the first feasible solution found has minimum x_1 and each successive solution has non-decreasing x_1. Whenever a feasible solution $(\hat{x}_1, \hat{x}_2)$ is found, the upper bound on x_2 can be set to $\hat{x}_2$ for all future solutions. This search will find the Pareto frontier. Furthermore, one of these points will minimize x_3.

4.3.7 Completion Bounds in the Navigation Level

The A* algorithm is an improvement on Dijkstra's well-known best-first search algorithm for shortest path problems with non-negative weights. The A* algorithm relies on a lower bound function, called the *heuristic function*, to estimate the remaining cost to-go at any state and to expand the search frontier only in the most promising direction (e.g., Russell and Norvig 2020).

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 $(L_j, t_j) \in \mathcal{V}^{\text{nav}}$ using A*, the heuristic function $h((l, t), (L_j, t_j))$ must give a lower bound on the remaining time/contention penalties to-go from any vertex $(l, t) \in \mathcal{V}^{\text{nav}}$ to the goal (L_j, t_j).

A heuristic function for contention penalties is difficult to define because the contention penalties can appear on arbitrary edges of the time-expanded navigation graph, giving rise to a time-dependent heuristic function. The trivial heuristic function $h_{\text{pen}}((l, t), (L_j, t_j)) := 0$ is used instead.

A heuristic function for time $h_{\text{time}}((l, t), (L_j, t_j))$ can be computed as follows. Observe that a time-independent heuristic function $h_{\text{time}}(l, L_j)$ is always a lower bound on the equivalent time-dependent heuristic because the time-dependent heuristic can give a higher lower bound due to waiting, which is not possible in the time-independent heuristic. Therefore, the time-independent heuristic is a weaker but still valid lower bound and $h_{\text{time}}((l, t), (L_j, t_j)) := h_{\text{time}}(l, L_j)$ can be defined to be time-independent.

To pre-compute $h_{\text{time}}(l, L_j)$ (i.e., the minimum number of steps from any location $l \in \mathcal{L}$ to L_j), a preliminary A* algorithm is run on a non-time-expanded graph whose vertices are the locations $\mathcal{L}$. This A* algorithm is run using the Manhattan distance as its heuristic. The costs resulting from this computation is then stored as the heuristic $h_{\text{time}}(l, L_j)$ for the A* search on the time-expanded network $\mathcal{G}^{\text{nav}}$.

4.3.8 Pseudocode

Algorithm 1 presents the pricer. A *label* is a 5-tuple that contains a partial sequence, the corresponding partial path, the accumulation of reduced costs, the arrival time and the accumulation of contention penalties. 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

```
1  Function Pricer((I_i)_{i∈V^seq}, π_a, (ρ_j)_{j∈R^+}, (σ_b)_{b∈B}):
       input : the time ranges of all vertices (I_i)_{i∈V^seq}, the dual solution π_a of Constraint (1b) for agent a, the
               dual solutions (ρ_j)_{j∈R^+} of Constraint (1c), the dual solutions (σ_b)_{b∈B} of Constraint (4)
       output: a set of sequence-path pairs with negative reduced cost
2      forall i ∈ V^seq do
3          labels_i ← NewSet()
4      open ← NewPriorityQueue()
5      open.Add(((⊤), ((L_⊤, 0)), −π_a, 0, 0))
6      while ¬open.IsEmpty() do
7          ((⊤, r_1, ..., r_n), ((l_0, 0), ..., (l_t, t)), c̄, t, z) ← open.Pop()
8          forall j ∈ V^seq : (r_n, j) ∈ E^seq do
9              (⊤, r'_1, ..., r'_n) ← (⊤, r_1, ..., r_n, j)
10             paths ← NewSet()
11             forall (t̲_j, t̄_j) ∈ I_j do
12                 paths ← paths ∪ A*(((l_0, 0), ..., (l_t, t)), c̄, t, z, L_j, [t̲_j, t̄_j], (σ_b)_{b∈B})
13             forall (((l_0, 0), ..., (l_t', t')), c̄', t', z') ∈ paths do
14                 if j ∈ R^+ then
15                     c̄' ← c̄' − ρ_j
16                 if ¬Dominated((c̄', t', z'), labels_j) then
17                     labels_j ← labels_j ∪ {((⊤, r'_1, ..., r'_n), ((l_0, 0), ..., (l_t', t')), c̄', t', z')}
18                     open.Push(((⊤, r'_1, ..., r'_n), ((l_0, 0), ..., (l_t', t')), c̄', t', z'))
19     return {((⊤, r_1, ..., r_n, ⊥), ((l_0, 0), ..., (l_t, t)), c̄, t, z) ∈ labels_⊥ : c̄ < 0}
```

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

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 $-\pi_a$, arrival time 0 and 0 contention penalties. 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.

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. Preliminary experiments show that minimizing contention penalties and bounding time perform significantly better than vice versa. Line 3 initializes the time upper bound $\bar{t}$. Line 4 creates a map data structure that associates every vertex with a label, which represents a partial path, the accumulation of reduced costs, the current time and the accumulation of contention penalties. Line 5 creates the root label corresponding to the partial path $((l_0, 0), \ldots, (l_t, t))$ input from the high level search. Line 6 creates a priority queue of vertices for future processing. This priority queue prefers vertices whose partial path has lower contention penalties and breaks ties in favor of lower estimated arrival time and then higher current time (i.e., closer to arrival). Line 7 adds the current vertex into the priority queue with the contention penalties z, the estimated arrival time $t + h(l_t, L_j)$ and the current time t as values for the priority. Lines 8 to 10 loop over every label. Line 11 proceeds if the partial path respects the time upper bound. As the search progresses, the contention penalties increase and the latest arrival time decreases. If the partial path reaches the sink after the earliest possible arrival time (Line 12), Line 13 updates the time upper bound and Line 14 stores the new partial path.

Line 15 iterates over the outgoing edges. Lines 16 to 18 respectively compute the reduced cost $\bar{c}'$, the time t' and the contention penalties z' of the new partial path to l_{t+1}. If the new partial path reaches an unseen vertex (Line 19), Line 20 stores the new partial path and Line 21 adds the vertex to the priority queue. If the new partial path has lower contention penalties than the existing path ending at the same vertex (Line 22), Line 23 stores the new path and Line 24 replaces the priority in the priority queue. Line 25 returns the set of Pareto-optimal paths.

Note that the implementation does not strictly adhere to the pseudocode shown in Algorithms 1 and 2. Rather, the code has numerous enhancements to improve performance, such as storing a pointer to the parent label instead of copying the partial path when making an extension and storing the arrival time estimate $t + h(l_t, L_j)$ in the label instead of recomputing it repeatedly.

```
Function A*(((l_0,0),...,(l_t,t)), c̄, t, z, L_j, [t̲_j, t̄_j], (σ_b)_{b∈B}):
    input : a partial path ((l_0,0),...,(l_t,t)), reduced cost c̄, time t, contention penalties z, the destination
            location L_j, the destination time range [t̲_j, t̄_j], the dual solutions (σ_b)_{b∈B} of Constraint (4)
    output: a set of Pareto-optimal paths from (L_a^+, 0) to (L_j, t_j) where t_j ∈ [t̲_j, t̄_j]
    paths ← NewSet()
    t̄ ← t̄_j
    closed ← NewMap()
    closed[(l_t,t)] ← (((l_0,0),...,(l_t,t)), c̄, t, z)
    open ← NewPriorityQueue()
    open.Add((l_t,t), z, t + h(l_t, L_j), t)
    while ¬open.IsEmpty() do
        (l_t,t) ← open.Pop()
        (((l_0,0),...,(l_t,t)), c̄, t, z) ← closed[(l_t,t)]
        if t + h(l_t, L_j) ≤ t̄ then
            if l_t = L_j ∧ t ≥ t̲_j then
                t̄ ← t
                paths ← paths ∪ {(((l_0,0),...,(l_t,t)), c̄, t, z)}
            forall (l_{t+1}, t+1) ∈ V^nav : ((l_t,t),(l_{t+1},t+1)) ∈ E^nav do
                c̄' ← c̄ + 1 − Σ_{b∈B} β_b^{a,((l_t,t),(l_{t+1},t+1))} σ_b
                t' ← t + 1
                z' ← z − Σ_{b∈B} β_b^{a,((l_t,t),(l_{t+1},t+1))} σ_b
                if ¬closed.Contains((l_{t+1}, t+1)) then
                    closed[(l_{t+1},t+1)] ← (((l_0,0),...,(l_t,t),(l_{t+1},t+1)), c̄', t', z')
                    open.Push((l_{t+1},t+1), z', t' + h(l_{t+1}, L_j), t')
                else if z' < closed[(l_{t+1},t+1)].z then
                    closed[(l_{t+1},t+1)] ← (((l_0,0),...,(l_t,t),(l_{t+1},t+1)), c̄', t', z')
                    open.DecreasePriority((l_{t+1},t+1), z', t' + h(l_{t+1}, L_j), t')
    return paths
```

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

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 child nodes. The following three branching rules are used.

4.4.1 Branching on Edges in the Sequencing Graph

The first branching rule fixes the sequences by branching on the edges in the sequencing graph. This is a standard branching rule commonly seen in the VRP literature. It selects an edge $\bar{e} \in \mathcal{E}^{\text{seq}}$ from the sequencing graph that is fractionally used an agent $\bar{a} \in \mathcal{A}$, i.e.,

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

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

$$\sum_{(s,p) \in \Lambda_{\bar{a}}} \alpha_s^{\bar{e}} \lambda_{\bar{a},s,p} \leq 0 \tag{7a}$$

and

$$\sum_{(s,p) \in \Lambda_{\bar{a}}} \alpha_s^{\bar{e}} \lambda_{\bar{a},s,p} \geq 1. \tag{7b}$$

Constraint (7a) is added to the first child to forbid the agent from taking the edge. Constraint (7b) 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 if the agent traverses $\bar{e}$. This branching rule is called until all edges in the sequencing graph have integer value, thereby fixing the sequences of all agents.

4.4.2 Branching on Path Lengths

The second branching rule fixes the objective value by branching on the path lengths. This branching rule first computes the set

$$\bar{\mathcal{A}} = \{a \in \mathcal{A} : \exists (s_1, p_1), (s_2, p_2) \in \Lambda_a, \lambda_{a,s_1,p_1} > 0, \lambda_{a,s_2,p_2} > 0, c_{p_1} \neq c_{p_2}\}$$

of agents that are using paths with different costs. Next, the branching rule chooses an agent $\bar{a} \in \bar{\mathcal{A}}$ that is using a path $\bar{p}$ with lowest cost $c_{\bar{p}} = \bar{k} - 1$, i.e.,

$$(\bar{a}, \cdot, \bar{p}) = \underset{a \in \bar{\mathcal{A}}, (s,p) \in \Lambda_a}{\arg\min} \{c_p : \lambda_{a,s,p} > 0\}.$$

Two child nodes are then created. In the first child, agent $\bar{a}$ can only use paths with length $\bar{k}$ or shorter. In the second child, agent $\bar{a}$ can only use paths with length greater than $\bar{k}$. These constraints are enforced by removing all incompatible paths from the master problem and tightening the time window of the sink $\perp \in \mathcal{V}^{\text{seq}}$ when pricing $\bar{a}$. This branching rule is called until all paths used by every agent have identical arrival time at its end location, and consequently, all feasible solutions in the subtree below have identical costs, providing a strong dual bound. This branching rule is previously used in MAPF (Lam et al. 2019, 2022) and is similar to branching on time windows in VRPs (Gélinas et al. 1995, Costa, Contardo, and Desaulniers 2019).

4.4.3 Branching on Vertices in the Navigation Graph

The third branching rule fixes the paths by branching on the vertices in the navigation graph. This branching rule selects a vertex $\bar{v}^{\text{nav}} \in \mathcal{V}^{\text{nav}}$ of the navigation graph that is fractionally used by an agent $\bar{a} \in \mathcal{A}$, i.e.,

$$0 < \sum_{(s,p) \in \Lambda_{\bar{a}}} \alpha_p^{\bar{v}^{\text{nav}}} \lambda_{\bar{a},s,p} < 1.$$

Next, the branching rule selects an edge $\bar{e}^{\text{seq}} = (i, j) \in \mathcal{E}^{\text{seq}}$ of the sequencing graph where $\bar{v}^{\text{nav}}$ appears. That is, between completing requests i and j, the agent visits $\bar{v}^{\text{nav}}$, which is used by more than one agent. The branching rule then creates three child nodes.

The first child node requires the agent to traverse $\bar{e}^{\text{seq}}$ and visit $\bar{v}^{\text{nav}}$ between i and j. This is imposed by adding one of Constraint (7b) if $\bar{e}^{\text{seq}}$ has not yet been branched on and forcing the navigation path to visit $\bar{v}^{\text{nav}}$ when extending a path from i to j. Making this branching decision several times deep in the branch-and-bound tree means that the agent needs to visit several $\bar{v}^{\text{seq}}$ between requests i and j. Since $\bar{v}^{\text{seq}}$ is time-indexed, the agent is simply forced to detour through all the $\bar{v}^{\text{seq}}$ in increasing time. In the code, this path with detours is found by calling the A* algorithm to find path segments from point to point.

The second child node also requires the agent to traverse $\bar{e}^{\text{seq}}$ but stops it from visiting $\bar{v}^{\text{nav}}$ when extending a sequence from i to j. In the code, this is imposed by removing $\bar{v}^{\text{nav}}$ from the navigation graph. The third child node prevents the agent from traversing $\bar{e}^{\text{seq}}$ using Constraint (7a). If decisions about the sequencing edge are incompatible with earlier decisions, then the new node is infeasible and is not considered further. The three child nodes partition the search space so that any feasible solution will appear in exactly one subtree. This branching rule was previously used in MAPF (Lam et al. 2019, 2022).

While it is possible to branch solely on the navigation vertex $\bar{v}^{\text{nav}}$ without stipulating during which sequencing edge $\bar{e}^{\text{seq}}$ the visit to $\bar{v}^{\text{nav}}$ 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 $\bar{v}^{\text{nav}}$ in every extension of a sequence. As the number of these branching constraints increase deep into the branch-and-bound tree, the pricing problem must choose whether to visit or avoid each $\bar{v}^{\text{nav}}$ 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.

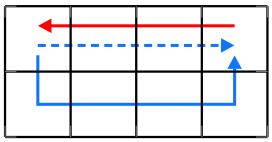

Figure 4: The blue agent cannot take its shortest path and detours to avoid colliding into the red agent, incurring an additional 2 cost above their shortest path costs of $3 + 3 = 6$.

5 BCPB-MAPD

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

The intuition behind BCPB-MAPD is explained using Figure 4. Two agents are attempting to cross the map. The total cost is 6 if both agents can take their shortest path. However, the blue agent detours to avoid colliding into the red agent and incurs an extra cost of 2, resulting in a total cost of 8. The main idea behind this algorithm is to first assign requests to the agents, assuming that they can take their shortest paths, and then find the costs of the detours necessary for avoiding collisions.

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 every 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 under-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. The feasibility and optimality of paths are enforced via Benders cuts.

Define Ψ_a as the set of request sequences for agent $a \in \mathcal{A}$. Associate every sequence $s \in \Psi_a$ with a cost $c_s \in \mathbb{Z}_+$. Let $\psi_{a,s} \in [0, 1]$ be a decision variable representing the proportion of selecting $s \in \Psi_a$. The variable $\theta \in \mathbb{R}_+$ represents the extra detour costs due to collisions in the path finding. The master problem begins as the linear program:

$$\min \sum_{a \in \mathcal{A}} \sum_{s \in \Psi_a} c_s \psi_{a,s} + \theta \tag{8a}$$

subject to

$$\sum_{s \in \Psi_a} \psi_{a,s} = 1 \qquad \forall a \in \mathcal{A}, \tag{8b}$$

$$\sum_{a \in \mathcal{A}} \sum_{s \in \Psi_a} \alpha_s^r \psi_{a,s} = 1 \qquad \forall r \in \mathcal{R}^+, \tag{8c}$$

$$\psi_{a,s} \geq 0 \qquad \forall a \in \mathcal{A}, s \in \Psi_a, \tag{8d}$$

$$\theta \geq 0. \tag{8e}$$

Objective Function (8a) minimizes the cost estimated by the PDPTW plus additional detour costs due to collisions. Constraints (8b) and (8c) are analogous to Constraints (1b) and (1c). Constraints (8d) and (8e) define the variable domains. The master problem can be recognized as the set partitioning master problem commonly seen in VRPs plus the θ term.

5.2 Generating Sequences

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

reduced cost and a time resource for upholding time windows. Collisions and detour costs/time are not handled in pricing, but rather, using Benders cuts in the master problem.

The pricer first pre-computes a cost matrix $d_{i,j} = h(L_i, L_j)$ corresponding to a lower bound on the distance between any two vertices $i, j \in \mathcal{V}^{\text{seq}}$ by calling an A* algorithm to determine the distance of the shortest path between L_i and L_j, as in Section 4.3.7. The pricer then executes a standard labeling algorithm to find a sequence with negative reduced cost, as if the cost under-estimates $d_{i,j}$ are valid. A partial sequence starting and ending at $\top$ is extended outward towards $\bot$. 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 (8c). 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 $\bot$ has negative reduced cost, it is added to Ψ_a together with a new variable.

5.3 Resolving Conflicts

Whenever an integer feasible solution to the master problem is found, a MAPF-like 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

$$w_a^e = \sum_{s \in \Psi_a} \alpha_s^e \psi_{a,s} \tag{9}$$

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

$$\mathcal{W} = \{(a, e) \in \mathcal{A} \times \mathcal{E}^{\text{seq}} : w_a^e = 1\} \tag{10}$$

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

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

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

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

where $\delta \in \mathbb{Z}_+$ is the difference between the cost of the PDPTW solution and the MAPF solution. Notice that both the feasibility cuts and the optimality cuts are robust and hence will not impact the structure of the pricing problem. 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-like 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.

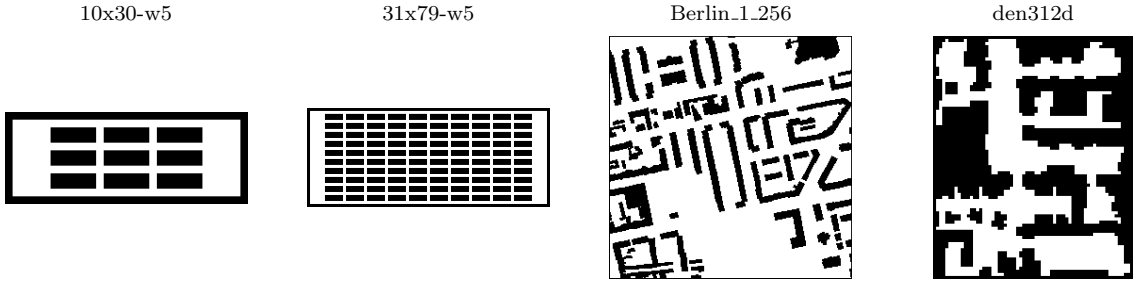

Figure 5: Four maps used in the experiments.

	BCP-MAPD	BCPB-MAPD	Two-Stage
Optimal	419 (26.2%)	427 (26.7%)	62 (3.9%)
Feasible	493 (30.8%)	1,191 (74.4%)	1,183 (73.9%)
Average gap (all instances)	69.3%	30.8%	30.1%
Average gap (432 instances)	0.2%	4.2%	4.6%
Arithmetic mean time (seconds)	2,868.3	2,821.2	1,522.5
Geometric mean time (seconds)	1,583.2	1,713.6	285.5

Table 1: Summary statistics of the results.

6 Experiments

The experiments compare BCP-MAPD and BCPB-MAPD against a baseline two-stage heuristic that first solves the PDPTW and then uses the sequences in the optimal PDPTW solution to solve a MAPF-like path finding problem. Note that this two-stage heuristic can compute an optimality gap because its first stage is a relaxation, providing a lower bound, and its second stage finds feasible solutions, providing an upper bound. 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 9.0.0 for branch-and-bound and Gurobi 11.0.1 for the linear programming restricted master problem.

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 a time window. Given the finite time windows and a finite number of agents, some instances could be infeasible.

Two warehouse maps (Li et al. 2019a, 2020), one city map (Stern et al. 2019) and one computer game map (Stern et al. 2019) are chosen. The maps are shown in Figure 5. The first and second maps 10x30-w5 and 31x79-w5 are modeled on automated warehouses, which contain narrow single-lane corridors. The third map Berlin_1_256 is a medium-size map that contains large open spaces for agents to wander and several choke points where congestion could occur. The fourth map den312d is a small map with vast open spaces but does not contain narrow choke points.

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 Intel Xeon Gold 6338 CPU with 500 GB of main memory.

6.1 Comparing the Algorithms

Table 1 reports overall statistics for the three algorithms. BCP-MAPD, BCPB-MAPD and the two-stage heuristic respectively find optimal solutions to 26.2%, 26.7% and 3.9% of the 1,600 instances and feasible solutions to 30.8%, 74.4% and 73.9% of the instances. None of the instances are proven to be infeasible (although none of them are confirmed to be infeasible either). Overall, BCPB-MAPD finds more optimal solutions than BCP-MAPD and more feasible solutions than the two-stage heuristic, indicating that it is better suited to the chosen instance set.

Averaging over all instances, BCP-MAPD, BCPB-MAPD and the two-stage heuristic find an average optimality gap of 69.3%, 30.8% and 30.1%, with a value of 100% assumed for instances where a gap is

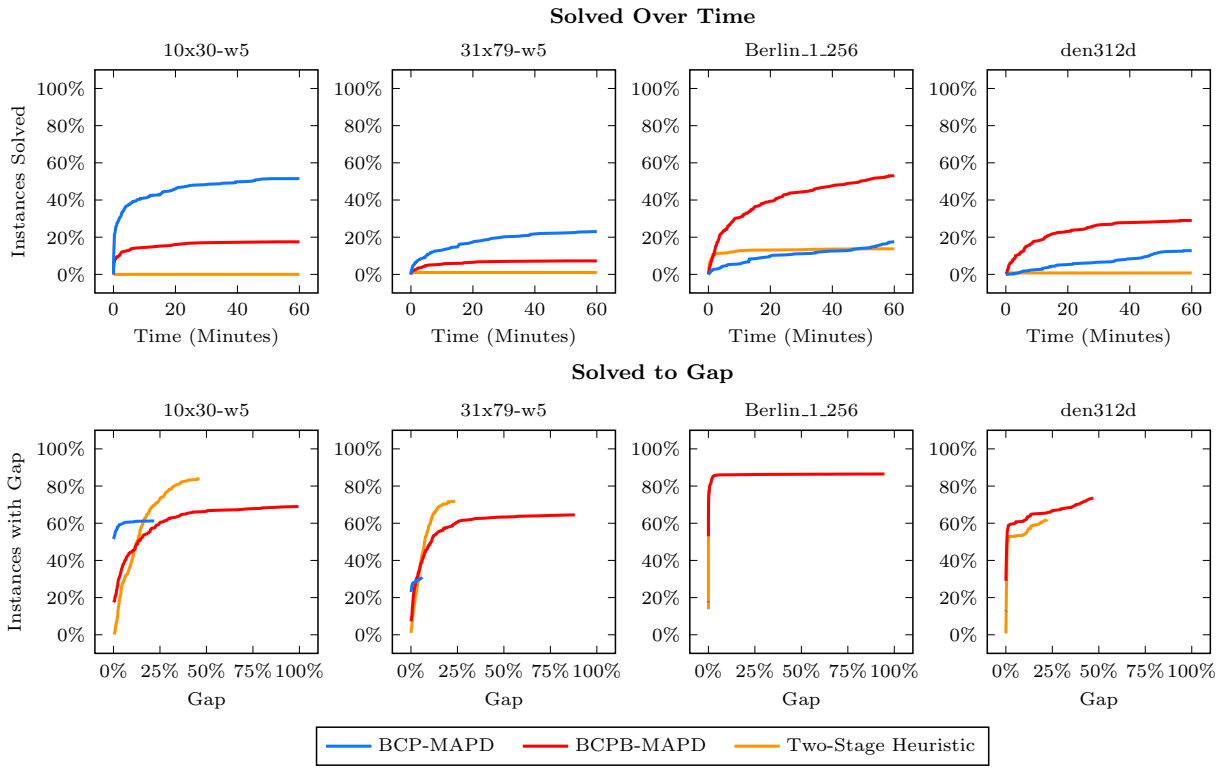

Figure 6: Percentage of instances solved over time for each map (higher is better) and percentage of instances solved up to an optimality gap (higher is better).

not available. This default value skews the result for BCP-MAPD given that it finds significantly fewer feasible solutions than BCPB-MAPD. There are 432 instances for which all algorithms find an optimality gap. On these instances, the average gaps are 0.2%, 4.2% and 4.6% respectively. Taken over all instances, the average gap by BCP-MAPD is more than double that of BCPB-MAPD. Averaged over the same subset of instances, BCP-MAPD achieves more than an order of magnitude smaller gaps.

The arithmetic mean solve times are 2,868.3, 2,821.2 and 1,522.5 seconds and the geometric mean solve times are 1,583.2, 1,713.6 and 285.5 seconds, where timed out instances are counted as 3,600 seconds. Just like the average gap results, the ranking of BCP-MAPD versus BCPB-MAPD are swapped depending on the statistic.

Even though BCP-MAPD and BCPB-MAPD solve a similar number of instances, their performance are highly dependent on the layout of obstacles. The first row in Figure 6 shows the percentage of instances solved optimally over time for each map. The second row shows the percentage of instances solved up to any given optimality gap.

The obstacles in 10x30-w5 represent single-lane corridors, which heavily impede the agents and cause severe congestion. Consequently, the PDPTW relaxation gives a poor lower bound and the two-stage heuristic completely fails to prove optimality or infeasibility on any instance. BCPB-MAPD performs better, closing 17.5% of the small warehouse instances. In contrast, the joint optimization in BCP-MAPD is vastly superior, solving 51.5% of these instances. At time out, BCP-MAPD finds optimality gaps of up to 21.8% for 61.3% of the 400 10x30-w5 warehouse instances. It fails to find either an upper bound or lower bound to the remaining instances. BCPB-MAPD uses a much easier relaxation and hence is able to find gaps for 69.0% of these instances, whose gaps range up to 99.5%. Note that a gap of 100% indicates that the upper bound is twice as costly as the lower bound but it is by no means a limit since optimality gaps can extend to infinity.

On the second warehouse map 31x79-w9, the lower number of instances solved by all algorithms reflects its larger size but the overall performance characteristics of the three algorithms are similar to the trends seen in the smaller warehouse map. The joint optimization of BCP-MAPD is clearly beneficial on highly congested instances. Even though BCP-MAPD solves more instances than BCPB-MAPD, the simpler relaxation in BCPB-MAPD enables it to obtain finite gaps (up to 88.1%) for 64.5% of the instances. In contrast, BCP-MAPD could only find gaps of up to 6.0% to 31% of the instances.

The performance of BCP-MAPD and BCPB-MAPD are reversed on the third map Berlin_1_256 compared to the warehouse maps. The time spent by BCP-MAPD to solve the navigation problem is not productive because the agents seldom interact given the availability of space and therefore the VRP relaxation provides an adequate lower bound. For the same reasons, the two-stage heuristic proves optimality for 13.8% of the instances and even solves more instances than BCP-MAPD until near the time limit. BCP-MAPD achieves gaps of up to 0.01% for 18% of the instances, essentially optimizing an instance or timing out with no further information. BCPB-MAPD finds gaps of up to 94.7% for 86.5% of the instances. Curiously, the two-stage heuristic terminates with gaps of up to 0.4% for 78.0% of the instances, unquestionably demonstrating the lack of difficulty in these instances.

The trends seen in Berlin_1_256 are also repeated in the fourth map den312d due to the large open spaces. BCPB-MAPD outperforms BCP-MAPD because the VRP relaxation is strong when conflicts seldom occur and hence it can avoid solving the navigation problem repeatedly. In the absence of conflicts, the VRP relaxation is tight, which allows the two-stage heuristic to find provably optimal solutions to three of the smaller instances. The optimality gap results corroborate this finding. BCPB-MAPD and the two-stage heuristic respectively find gaps of up to 47.0% for 73.5% of the instances and up to 22.2% for 62.0% of the instances. BCP-MAPD displays considerable difficult at these large but easy instances, achieving gaps of up to 0.3% for 13.0% of the instances, again solving an instance or terminating with unknown status.

Figure 7 shows the percentage of instances solved optimally by the three solvers on each map, separated by number of agents and orders. On the small warehouse map 10x30-w5, high congestion compromises the VRP relaxation, causing the two-stage heuristic to fail regardless of size. BCPB-MAPD, which is based on similar two-stage ideas, also deteriorates very quickly as the number of agents grows. Meanwhile, the simultaneous optimization in BCP-MAPD enables it to scale significantly better, dominating the other two algorithms.

Broadly similar results are obtained for the larger warehouse map 31x79-w5. The two-stage heuristic could prove optimality on four of the twenty instances with 20 agents. Similar to the smaller warehouse, BCPB-MAPD fails at scaling beyond the small instances and is almost dominated by BCP-MAPD at all instance sizes.

On Berlin_1_256, BCPB-MAPD holds steady up to 80 agents and then its performance collapses. Nevertheless it still dominates BCP-MAPD at all instance sizes. The two-stage heuristic finds optimal solutions even with up to 100 agents and 80 orders, indicating that the VRP relaxation is strong on this sparse map. On den312d, the distinction between BCP-MAPD and BCPB-MAPD is less pronounced but BCPB-MAPD still clearly dominates.

Figure 8 shows the average difference between the suboptimal objective value found by the two-stage heuristic and the optimal objective value found by either BCP-MAPD or BCPB-MAPD. There are 558 instances for which this result is available and the remaining instances are ignored. The two-stage heuristic often finds near optimal solutions whenever an optimal solution is available by other means. Admittedly, these are the easier instances since they are optimally solvable within the time limit. Nonetheless, this analysis demonstrates that the two-stage heuristic performs remarkably well, presumably due to the grid structure of the maps because 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).

The overall findings are intuitive. 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. The joint optimization in BCP-MAPD is expensive but advantageous when the agents interact frequently or are gridlocked because simultaneously reasoning about the task assignment and path finding is critical to obtaining a strong lower bound and feasible solutions. In contrast, the partially sequential optimization in BCPB-MAPD is preferable when agents seldom encounter each other and crowding is less widespread because fewer calls to the path finding solver are required. This key finding is clearly demonstrated on the large but easy maps, where the MAPD linear relaxation in BCP-MAPD proves intractable but the easier PDPTW discrete relaxation in BCPB-MAPD guides it toward poor-quality but feasible solutions. It is also fascinating that the two-stage heuristic obtains nearly identical optimality gaps as BCPB-MAPD in many of these large but easy instances, demonstrating that independent sequential optimizations are more than adequate when agents interact infrequently.

Comparing against the state of the art is difficult due to the differences in problem variants scattered across the literature. The closest works are CBSS and the baseline A* algorithm by Ren, Rathinam, and Choset (2023). Recall from Section 2 that CBSS is theoretically optimal but the implementation calls a heuristic subroutine and hence nullifies all optimality guarantees. Their experiments show that the

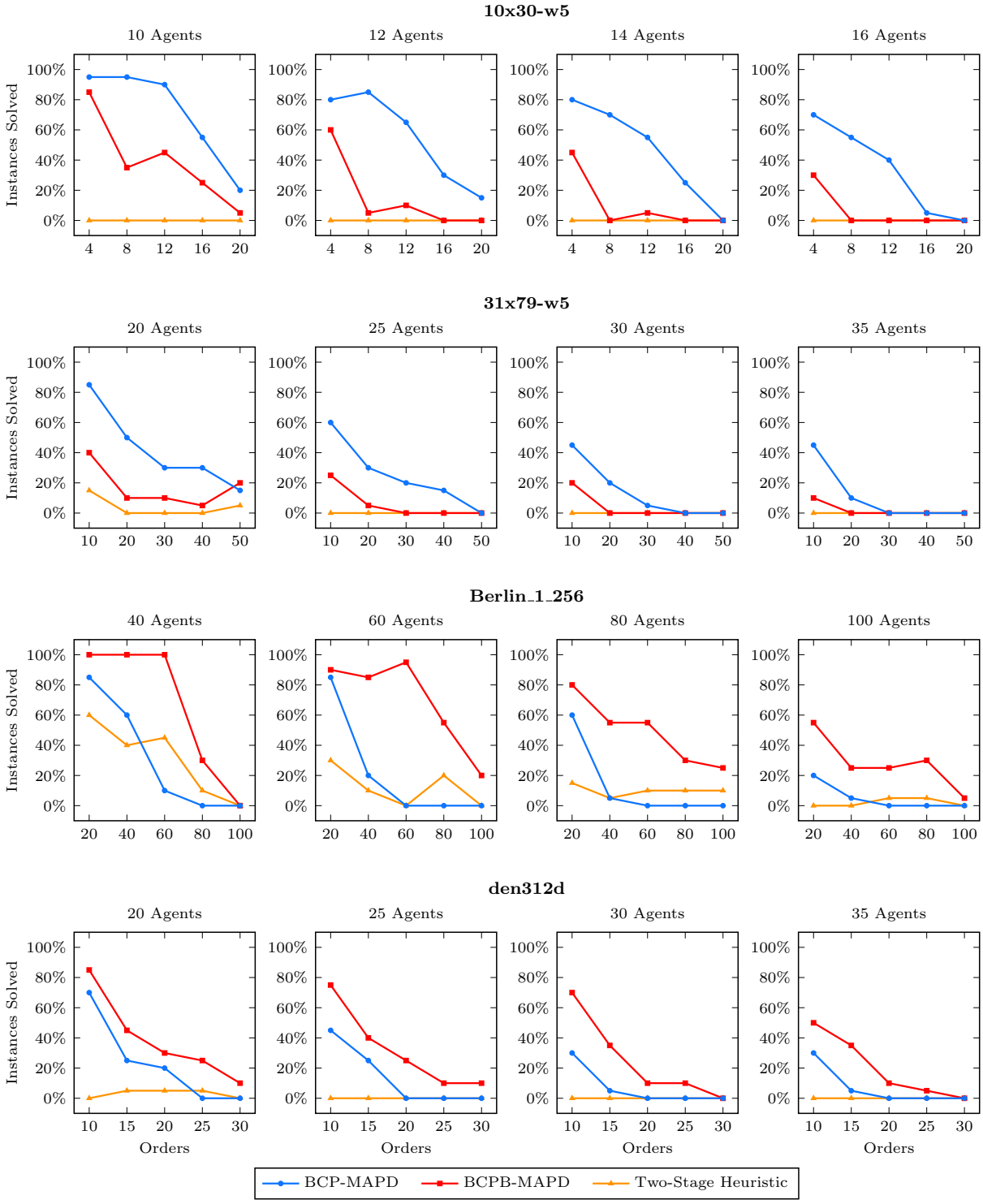

Figure 7: Percentage of instances solved optimally, plotted for different maps, number of agents and number of orders (pickup-delivery pairs). Higher is better.

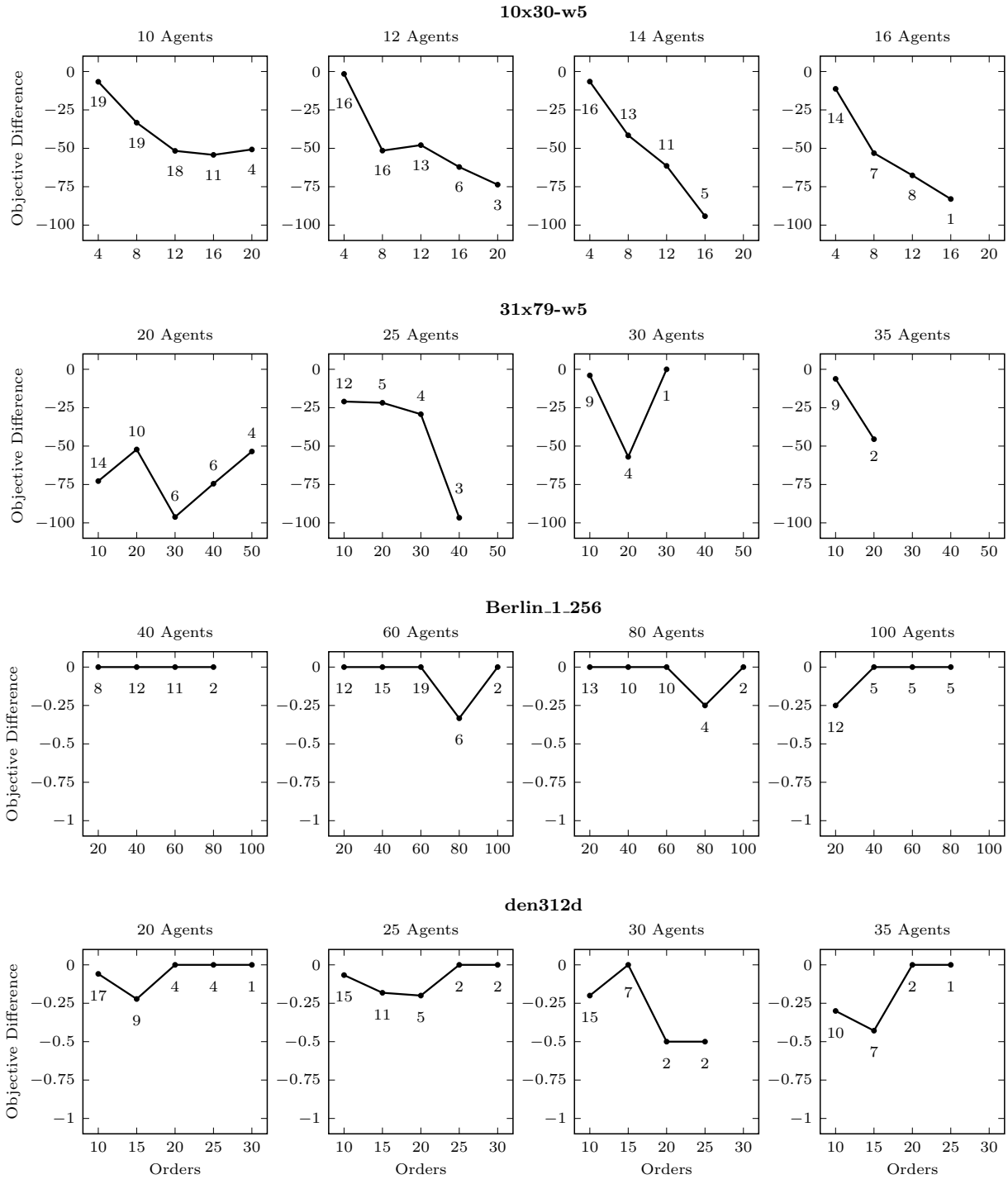

Figure 8: Average difference between the suboptimal objective value found by the two-stage heuristic and the optimal objective value found by either BCP-MAPD or BCPB-MAPD. Every data point is labeled with the number of instances averaged over.

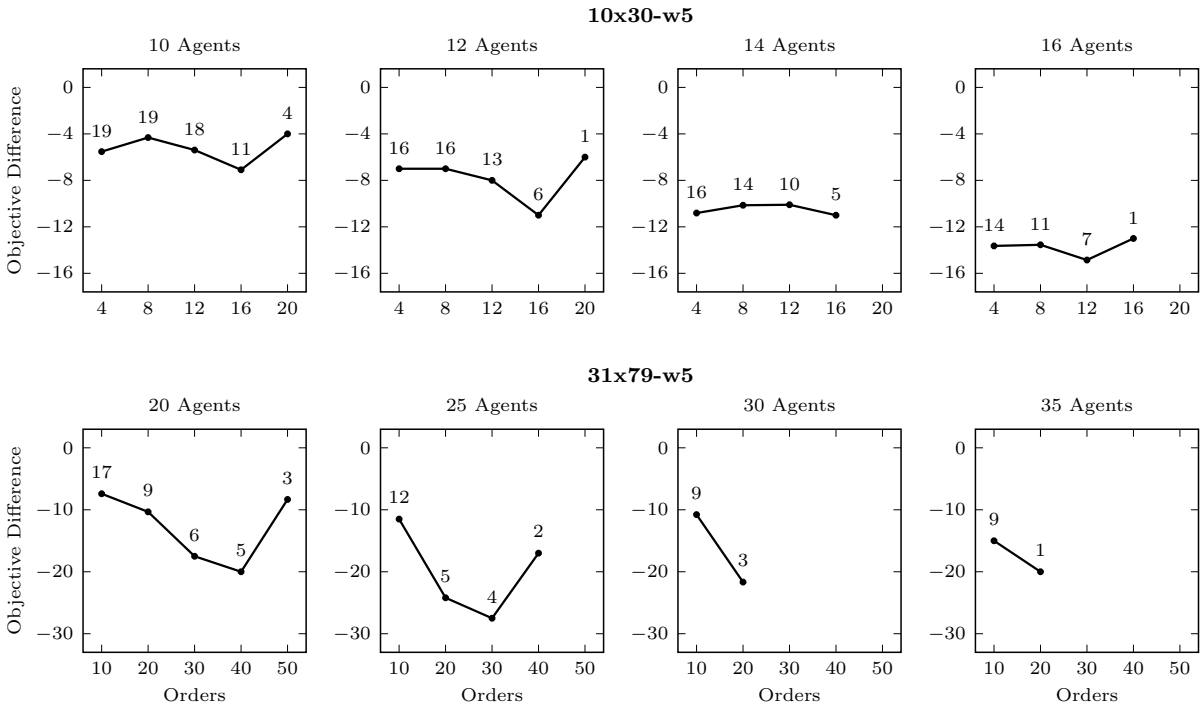

Figure 9: Average difference in cost after allowing agents to move through obstacles while not carrying an item. Every data point is labeled with the number of instances averaged over.

A* baseline fails to find any optimal solution even with 5 agents, while CBSS finds feasible solutions to instances with 20 agents and 50 requests. In comparison, BCP-MAPD proves optimality on instances of a similar size (20 agents and 50 orders) and BCPB-MAPD scales significantly higher on sparse maps, optimizing instances with 100 agents and 100 orders.

6.2 Allowing Empty Agents to Move Under Shelves

In automated warehouses, robots lift up shelves from underneath. While a robot is not carrying a shelf (i.e., is empty), 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. In the code, this is implemented by switching to an empty map when running A* on an empty agent.

Figure 9 shows the average difference in the optimal objective value on the 354 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 data point. Not many instances are 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. This is likely due to the path symmetries present in rectangular grid maps.

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 and algorithm 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. BCP-MAPD and BCPB-MAPD are believed to be the first two high-performance exact algorithms for MAPD.

Experimental results indicate that neither algorithm dominates. BCP-MAPD is advantageous in crowded environments where its ability to jointly reason about the task assignment and path finding is necessary. BCPB-MAPD is better suited to deserted environments where agents rarely encounter each other because its VRP relaxation provides a strong lower bound in conflict-free path finding. Using the same VRP relaxation as BCPB-MAPD, the two-stage sequence-first and navigate-second heuristic also performs unexpectedly well on sparse maps, obtaining provably optimal solutions even to a few large instances. Comparing qualitatively against the state-of-the-art CBSS, which cannot guarantee optimality, BCP-MAPD proves optimality to similarly-sized instances and BCPB-MAPD scales substantially higher on sparse maps, finding optimal solutions to instances with 100 agents and 100 pickup-delivery pairs.

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 grid structure of the maps.

One direction for future work is to develop an adaptive algorithm that dynamically selects either the joint or sequential optimization depending on the density of obstacles and the branching decisions, which could detour agents and turn a crowded region into a conflict-free organized flow.

Another research direction is to scale up exact and heuristic algorithms to a few hundred agents and orders. Recent algorithms for MAPF are starting to tackle these industrial-size instances, which were intractable just a few years ago. Furthermore, 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 the Discovery Early Career Researcher Award DE240100042 and Discovery Projects DP190100013 and DP200100025, and by a gift from Amazon. We would also like to thank the anonymous reviewers whose comments have improved this paper.

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.

de Aragao MP, Uchoa E, 2003 *Integer program reformulation for robust branch-and-cut-and-price algorithms. Mathematical Program in Rio: a conference in honour of Nelson Maculan*, 56–61.

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.

Gélinas S, Desrochers M, Desrosiers J, Solomon MM, 1995 *A new branching strategy for time constrained routing problems with application to backhauling. Annals of Operations Research* 61(1):91–109.

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.

Hoenig W, Kumar TK, Cohen L, Ma H, Xu H, Ayanian N, Koenig S, 2016 *Multi-agent path finding with kinematic constraints. Proceedings of the International Conference on Automated Planning and Scheduling* 26(1):477–485.

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, 2019 *Branch-and-cut-and-price for multi-agent pathfinding. Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1289–1296.

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, 2019a *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.

Li J, Surynek P, Felner A, Ma H, Kumar S, Koenig S, 2019b *Multi-agent path finding for large agents. Proceedings of the AAAI Conference on Artificial Intelligence (AAAI).*

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.

Russell S, Norvig P, 2020 *Artificial Intelligence: A Modern Approach* (Pearson), fourth edition.

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.