Exact Methods for Solving Traveling Salesman Problems with Pickup and Delivery in Real Time

Ryan J. O'Neil · Karla Hoffman

September 2, 2018

Abstract The Traveling Salesman Problem with Pickup and Delivery (TSPPD) describes the problem of finding a minimum cost path in which pickups precede their associated deliveries. The TSPPD is particularly important in the growing field of Dynamic Pickup and Delivery Problems (DPDP). These include the many-to-many Dial-A-Ride Problems (DARP) of companies such as Uber and Lyft, and meal delivery services provided by Grubhub. We examine exact methods for solving TSPPDs where orders from different pickup locations can be carried simultaneously in real-time applications, in which finding high quality solutions quickly (often in less than a second) is often more important that proving the optimality of such solutions. We consider enumeration, Constraint Programming (CP), Mixed Integer Programming (MIP), and hybrid methods combining CP and MIP. Our CP formulations examine multiple techniques for ensuring pickup and delivery precedence relationships. Finally, we attempt to provide guidance about which of these methods may be most appropriate for fast TSPPD solving given various time budgets and problem sizes.

Keywords traveling salesman problem; pickup and delivery; integer programming; constraint programming; computational experiments; hybrid optimization methods; real-time optimization

Ryan J. O'Neil
Systems Engineering & Operations Research Department
George Mason University, Fairfax, Virginia 22030
E-mail: roneil1@gmu.edu
Decision Engineering Department
Grubhub, Chicago, Illinois 60602
E-mail: roneil@grubhub.com

Karla Hoffman
Systems Engineering & Operations Research Department
George Mason University, Fairfax, Virginia 22030
E-mail: khoffman@gmu.edu

1 Introduction

The Traveling Salesman Problem (TSP) is the search for a minimum cost Hamiltonian circuit connecting a set of locations. In general, the TSP is NP-complete [34]. The first paper of note related to TSP research solved an instance of "49 cities, one in each of the 48 states and Washington, D.C." [10]. Algorithms now exist that, given sufficient time, can solve TSPs with tens of thousands of nodes, and its computational boundaries continue to be pushed forward [2].

The TSP is an important problem in inspiring algorithmic research, to the substantial benefit of the optimization community. It also provides a practical application in its own right, particularly in the context of Vehicle Routing Problems (VRP). VRPs are common in industry and involve routing a set of vehicles to visit a set of nodes at minimum cost. At its surface, the only substantive difference between the TSP and the VRP is the latter's use of multiple vehicles to service customer demand.

The Pickup and Delivery Problem (PDP) is an increasingly prevalent industrial form of the VRP in which each of a set of requests must be picked up from one or more locations and delivered to one or more delivery points. Pickups must precede their associated deliveries in any feasible route, and each pair must be serviced by the same vehicle. Pickup and delivery locations can be distinct to each request, as in the many-to-many Dial-A-Ride Problem (DARP) of Psaraftis (1980) [37] Bertsimas et al. (2018) [7] and the Meal Delivery Routing Problem (MDRP). Given multiple pickup and delivery

requests, the PDP seeks to route a set of vehicles to service those requests at minimum cost [38].

Solving a PDP with a single vehicle is equivalent to solving a TSP with precedence constraints on the pickup and delivery nodes. This form is referred to as the Traveling Salesman Problem with Pickup and Delivery (TSPPD) [12]. There is a great deal of literature on the TSP and some of its variants, such as the TSP with Time Windows (TSPTW) [2]. In contrast, there has been less attention paid to the TSPPD, despite its practical applicability [38]. TSPPDs and their variants play an increasingly important role in industrial routing applications. This importance is witnessed by a proliferation of ride hailing and sharing companies, as well as on-demand delivery service providers for everything from groceries, alcoholic beverages, and meals, to snacks and convenience store items.

Different forms of on-demand pickup and delivery have their own objectives, constraints, and levels of dynamism and urgency for decision making [28]. For instance, in dynamic DARP, end users are inconvenienced from the moment they request a ride. In the MDRP, time from request to delivery cannot be less than the restaurant preparation time and travel time from the restaurant to the diner.

1.1 Problems with Small Routes

The divide between research and industrial use of the TSP becomes clearly evident when one considers practical limitations associated with routing. For example, delivery trucks are often limited to routes of fewer than 100 stops due to physical considerations, such as vehicle capacity [9]. Routes used for high volume restaurant delivery are even shorter due to the perishability of goods, and typically involve fewer than 10 stops.

Even in the context of the broader PDP with many vehicles and many more nodes (e.g. hundreds or thousands), the allowable individual routes tend to be relatively short. Solving real-world delivery problems often requires solving, in a very short time frame (i.e. fractions of a second to not more than minutes), a huge numbers of problems each having a very small number of nodes. And, often, resolving some subset of these problems as the delivery service learns of changing demand and/or service times.

This lack of attention reflects a common theme of exact approaches in the TSP optimization literature, which seek to optimize ever longer routes. The primary mechanism for finding exact solutions to TSPs is a branch-and-cut procedure based on Dantzig et al. (1954) [10] in which the TSP is relaxed into a two-matching problem, binary arc variables are relaxed to continuous variables with 0-1 bounds, and cuts are added to the formulation as infeasible solutions are found by the optimization. Thus the procedure begins with an infeasible solution and works toward feasibility. The bulk of the computation is based on getting good lower bounds on the problem, as good upper bounds are usually obtained through the Lin-Kernighan heuristic.

Branch-and-cut is by no means the only model for solving TSPs. One could consider adding cuts to the two-matching or assignment relaxations as pure integer solutions are found, as done in Pferschy and Staněk (2016) [36]. This approach may not be as effective in proving optimality, but it removes complexity and cutting planes, making its implementation more convenient. Such approaches retain the issue of starting infeasible and working toward feasibility. In either case, most procedures employ heuristics to obtain feasible solutions, thereby supplying upper bounds on the problem.

Infeasibility can be particularly painful in dynamic real-time logistics, where optimization procedures are time limited. Such problems may prefer obtaining a "good" solution quickly to an optimal solution eventually. In these circumstances, one might reasonably ask, "What is the cost of optimality?" Or the corollary, "What is the value of knowing one is likely to obtain a near-optimal solution within a certain amount of time?" For the class of problems that we are considering, time budgets for route optimization are on the order of milliseconds or, at most, seconds.

1.2 Hybrid Methods

Another issue with standard exact approaches to solving practical TSP instances is inflexibility. Industrial problems often have side constraints. Of particular interest here, the PDP requires that associated pickups and deliveries be serviced by the same vehicle and that the former precede the latter.

Other common constraints involve time windows, such as in the PDPTW [11]. For example, for meal delivery service operations, an order cannot be picked up from a restaurant before it is ready. A delivery cannot be made to a residence outside of an agreed-upon time window. Further, vehicles may have capacity constraints. This requirement is particularly important if bicycle couriers are used, as they have limited carrying capacity. These are just a few of the potential side constraints that may be required in an industrial model for either physical or business reasons.

Milano et al. (2001) [30] characterize the strengths and weaknesses of Mixed Integer Programming (MIP)

and Constraint Programming (CP) in terms of optimality and feasibility reasoning. MIP excels at proving optimality by working to improve both the upper and the lower bounds, but may take longer to obtain a feasible solution. The slowness in finding a first feasible solution is often because algebraic (linear) formulations fail to encode the special structure of a problem without requiring the use of "big M's" to indicate either-or syntax. Thus, side constraints such as precedence constraints or time windows that occur in TSP related to vehicle routing can be challenging for a MIP solver, but they adapt quite naturally to CP, since CP solvers can handle them directly in their propagation and search strategies [9, 35].

On the other hand, CP is much less efficient at optimality reasoning than MIP. This is partly due to poor propagation on objective functions, and partly due to a lack of built-in concepts around dual bounds and reduced-cost variable fixing. We address these difficulties by studying both MIP and CP forms of the TSPPD, then integrating them into hybrid models that combine the benefits of the two, similar to the models proposed in Focacci et al. (1999) [17] and Focacci et al. (2002) [16].

We test these models using problem instances generated from real-world meal delivery data. These are built from actual pickup and delivery locations observed at Grubhub, along with expected travel times connecting location pairs. We use a variety of problem sizes to characterize the performance of enumerative, MIP, CP, and hybrid models, and attempt to provide advice regarding which techniques are most appropriate for real-time optimization given varying time budgets and problem sizes.

Contributions of this paper include extensive computational experiments using these models for solving TSPPD instances, along with guidance for which techniques work best quickly and how they can be hybridized effectively. Our test set and source code for all model implementations are available for continued experimentation [20, 32]. Further, in the CP model implementation, we compare different forms of precedence that operate on circuit constraints, and provide a complete description of a propagator that enforces precedence without the use of time windows. Finally, we explore different branching forms for CP applied to the TSPPD.

2 Background

The Traveling Salesman Problem with Pickup and Delivery (TSPPD) is a modification of the Traveling Salesman Problem (TSP) that includes side constraints enforcing precedence among pickup and delivery node pairs. Each of n requests has a pickup node and a delivery node, and its pickup must occur before its delivery for a route to be feasible. The objective of the problem is to minimize total distance traveled while visiting each node exactly once. The TSPPD is formally described in Ruland and Rodin (1997) [38] and Dumitrescu et al. (2010) [12].

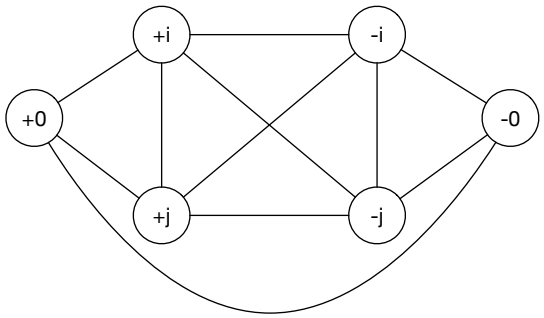

Fig. 1 Example TSPPD graph structure.

2.1 The TSPPD Polytope

The TSPPD is defined on an ordered set of pickup nodes $V_+ = \{+1, \ldots, +n\}$ and associated delivery nodes $V_- = \{-1, \ldots, -n\}$ such that $(+i, -i)$ form a request and $+i$ must precede $-i$ in a feasible route. V is defined as the union of V_+ and V_- with the addition of origin and destination nodes $\{+0, -0\}$. $E_\pm$ is the set of edges connecting $V_+ \cup V_-$. E is the union of $E_\pm$ with all feasible edges connecting to the origin and destination nodes. The graph $G = (V, E)$ includes all nodes and edges required to describe the TSPPD.

$$V = \{+0, -0\} \cup V_+ \cup V_-$$
$$E = \{(+0, -0)\} \cup \{(+0, +i) \mid i \in V_+\}$$
$$\cup \{(-0, -i) \mid i \in V_-\} \cup E_\pm$$

For modeling convenience, the edge $(+0, -0)$ must be included in any feasible solution. The set of edges E is defined such that it includes only feasible edges. That is, it is not possible that a TSPPD route begins with a delivery or ends with a pickup. Figure 1 shows an example of the feasible edges for a TSPPD with two requests. Note that it is possible to use the same formulation to solve Hamiltonian paths with precedence rather than circuits by simply assigning a cost of zero to the $(+0, -0)$ edge.

2.1.1 The Symmetric TSPPD

In the case of the symmetric TSPPD (STSPPD), each edge $(i, j) \in E$ is the same as the edge (j, i) and has

the same costs and edge variables. c_{ij} specifies a non-negative cost for each edge $(i,j) \in E$. By convention, $c_{+0,-0} = 0$. $x_{ij} \in \{0,1\}$ is a binary decision variable for each $(i,j) \in E$ with the value $x_{ij} = 1$ if the edge (i,j) is in a solution and 0 otherwise. $\delta(S) = \{(i,j) \in E \mid i \in S, j \notin S\}$ is the cutset containing edges that connect $S \subset V$ and $\bar{S} \subset V$. For any node $i \in V$, $\delta(i) = \delta(\{i\})$. The STSPPD is defined using Formulation 1, as in Ruland and Rodin (1997) [38].

$$\min \quad \sum_{(i,j)\in E} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{s.t.} \quad & x_{+0,-0} = 1 && && (1)\\
& x(\delta(i)) = 2 && \forall\, i \in V && (2)\\
& x(\delta(S)) \geq 2 && \forall\, S \subset V && (3)\\
& x(\delta(S)) \geq 4 && \forall\, S \subset V, \{+0,-i\} \subset S, \\
& && \quad \{-0,+i\} \subset V \setminus S && (4)\\
& x_{ij} \in \{0,1\} && \forall\, (i,j) \in A &
\end{aligned}$$

Formulation 1: STSPPD as provided by Ruland and Rodin [38]

Constraint (1) requires that the edge connecting the origin and destination nodes be part of any feasible solution. The degree constraints (2) require that each node is entered and exited in all feasible routes, but by itself leaves open the possibility of disconnected subtours. Constraints (3) accomplish subtour elimination, forming a complete representation of the TSP. The final set of constraints (4) require that pickups occur in routes before their respective deliveries [38].

2.1.2 The Asymmetric TSPPD

The asymmetric TSP (ASTP) polytope can be similarly adapted to the asymmetric TSPPD (ATSPPD) by associating the x variables with arcs (i.e. directed edges) and replacing the two-degree constraints with assignment constraints. Instead of an edge set E we define an arc set A which contains only the feasible arcs for the ATSPPD.

$$\begin{aligned}
A = & \{(-0,+0)\}\\
& \cup \{(+0,+i) \mid +i \in V_+\}\\
& \cup \{(-i,-0) \mid -i \in V_-\}\\
& \cup \{(+i,j) \mid +i \in V_+,\ j \in (V_+ \cup V_-) \setminus \{+i\}\}\\
& \cup \{(-i,j) \mid -i \in V_+,\ j \in (V_+ \cup V_-) \setminus \{+i,-i\}\}
\end{aligned}$$

Formulation 2 is more general since it supports arc costs that are not the same bidirectionally. Further, it is intuitively satisfying to consider the TSPPD this way, as there is a natural asymmetry built into the structure of the problem: pickups must precede their associated deliveries. As in Formulation 1, the $x_{-0,+0}$ arc connecting the start and end nodes must be part of any feasible tour, $+0$ must connect to a pickup, and -0 must be preceded by a delivery. One small additional difference is that Formulation 2 does not include x variables for arcs starting at a delivery and ending at its associated pickup.

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{s.t.} \quad & \sum_{(i,j)\in A} x_{ij} = 1 && \forall\, i \in V && (5)\\
& \sum_{(i,j)\in A} x_{ij} = 1 && \forall\, j \in V && (6)\\
& x(\delta(S)) \geq 1 && \forall\, S \subset V && (7)\\
& x(\delta(S)) \geq 4 && \forall\, S \subset V, \{+0,-i\} \subset S, \\
& && \quad \{-0,+i\} \subset V \setminus S &\\
& x_{ij} \in \{0,1\} && \forall\, (i,j) \in A && (8)
\end{aligned}$$

Formulation 2: ATSPPD

In Formulation 2, constraints (5) and (6) require that each node directly precede and follow exactly one other node. Constraints (7) accomplish subtour elimination, while precedence is enforced using the same constraints as in Formulation 1.

2.2 Enumerative Methods

One reason solving TSPs is challenging is that the number of feasible solutions grows as a factorial function of the number of nodes. Yet enumeration may still be a valid technique for solving small TSPPDs. If it is guided by some information about the problem, enumeration may be able to discover good routes quickly without any overhead of model formulation.

In addition to more sophisticated models, we also consider an enumerative approach. Our purpose is to determine which size problems enumeration can successfully solve, and at which point sophisticated modeling techniques tend to more effectively solve the problem. Algorithm 1 uses a recursive function to search the feasible set of TSPPD routes, while tracking the best solution discovered at each point in the search tree.

This algorithm maintains a current node and a partial tour starting at $+0$. Every recursion loops over the arcs from the current node and appends each feasible one to the tour individually prior to recursing. If an arc's added cost puts a partial route's cost over that of

Algorithm 1 Enumerative TSPPD Solver

```
function Initialize
    for all i ∈ nodes do
        arcs(i) ← ordered {j | j ∈ nodes \ {+0, −0}, j ≠ i}
    end for
    tour ← (+0)
    best ← ∅
end function

function Enumerate
    n1 ← last node in tour
    for all n2 ∈ arcs(current) do
        if n2 ∈ tour then
            continue
        else if cost(tour) + cost(n1, n2) ≥ cost(best) then
            continue
        else if n2 ∈ V− and pickup(n2) ∉ tour then
            continue
        end if

        tour ← tour + next
        if |tour| ≥ n − 1 then
            c ← cost(tour) + cost(n2, −0)
            if best = ∅ or c < cost(best) then
                best ← tour + − 0
            end if
        else
            Enumerate()
        end if
        tour ← tour − next
    end for
end function

Initialize()
Enumerate()
```

the best known tour, that arc is ignored and its section of the search tree effectively fathomed. If a complete tour improves the best known solution, it is stored as the new incumbent.

The key to effective search and early discovery of good solutions is the ordering technique for arcs. During initialization, each node is assigned a sorted vector of next nodes. We use ascending arc cost as our sorting function, and others can be easily incorporated. Nodes that are infeasible for the current partial route are skipped.

A symmetric TSP instance with n pairs of nodes has $\#\mathrm{TSP}(n) = \frac{1}{2}(2n-1)!$ unique solutions [26], while a TSPPD of the same size has $\#\mathrm{TSPPD}(n) = \frac{(2n)!}{2^n}$ [38]. The size of the feasible set of the TSPPD grows slower with respect to the number of node pairs than that of the TSP. Obviously, the set of feasible solutions of the TSPPD is a subset of the feasible solutions to the TSP. Allowing only pickup and delivery node pairs in the route with precedence relations reduces the size of the TSP feasible set by $\frac{1}{2^{n-1}}$ [38].

With fewer feasible solutions, the worst case complexity of solving a TSPPD instance via enumeration is technically better than the TSP. As side constraints such as time windows or capacity constraints are added, the size of the feasible set reduces further. However, that size may no longer have a closed form solution [6].

As we shall see in the next sections, the classical technique of solving a TSP through relaxation of constraints can yield infeasible solutions, requiring the addition of more constraints to the model during solution. As more side constraints are added to the model, they can complicate the search or model structure. Branch-and-cut approaches relax the integrality constraints and work to tighten the Linear Programming relaxation by adding cuts and by restricting the range of the values for the integer values. In contrast, Constraint Programming, works directly on the original model and uses logical implications within a tree-search to tighten the feasible range for the integer variables.

2.3 Mixed Integer Programming Methods

While formulations 1 and 2 give complete representations of the STSPPD and ATSPPD polytopes, the number of subtour elimination and precedence constraints grow quickly as a function of the number of nodes. Most exact approaches to the TSP and its variants solve a relaxation of the problem that omits these constraints, and add constraints to the representation as they are violated by new solutions. This is embodied in a methodology called branch-and-cut, which relaxes the subtour elimination constraints as well as the integrality constraints as in Dantzig et al. (1954) [10]. Padberg and Rinaldi (1991) [33] refine the technique by using additional polyhedral cuts and including the Lin-Kernighan heuristic [27]. This general approach is the primary one used to prove optimality for large TSPs over the last several decades.

The primary motivation for the study presented here is obtaining very good solutions in milliseconds to relatively small TSPPD instances. Branch-and-cut methods have disadvantages in this domain. They are challenging to implement when adding subtour elimination constraints (SEC) as well as combs, ladders, and other constraints including those described in Dumitrescu et al. (2010) [12], and therefore require a great deal of modeling overhead.

2.3.1 Combinatorial Relaxations

Instead of beginning our search in fractional space, it may make more sense to adopt an approach similar to that of Pferschy and Staněk (2016) [36] and Aguayo et al. (2018) [1], and only consider integer-feasible solutions to two-matching or assignment models. In this

approach we solve a combinatorial relaxation containing only the degree constraints. SEC are added within the branch-and-bound search tree as they are violated. Such constraints are implemented as "lazy" constraints within the solver, which are allowed to cut off integer-feasible solutions and are checked whenever a new candidate solution is found.

We take the same approach with the TSPPD: solve a relaxation of the problem using only degree constraints (2) for the STSPPD, or constraint sets (5) and (6) for the ATSPPD. When a candidate solution is found, check to see if it contains subtours or violates precedence. If so, add SEC and precedence constraints as appropriate. Once a tour covering all nodes and satisfying presence constraints is found, that tour is optimal.

As integer-feasible solutions are discovered in the branch-and-bound tree, they are scanned for subtours. $\mathcal{T}$ is the set containing all sets of nodes in a tour in the current solution. If $\mathcal{T} > 1$, then for each set of nodes corresponding to a subtour $S \in \mathcal{T}$, we add a lazy constraint of either the form (3) or (7), eliminating that subtour from future solutions.

Constraints (3) or (7) remove a given subtour from the final solution. Thereafter, not all edges in the subtour can exist in any new solution. An alternative and equivalent formulation of the constraint is shown as (9) below. We call constraints (3) and (7) the "cutset" form and constraint (9) the "subtour" form, respectively.

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \tag{9}$$

The different SEC formulations achieve the same end, but impact different sets of edges. In particular, constraint set (9) includes not only the edges found in a disconnected subtour, but all edges connecting the nodes in the subtour S. This inclusion of additional edges makes the constraint stronger, and follows from the observation that only $|S| - 1$ edges from a subtour can be in a Hamiltonian circuit, and that circuit is comprised of n edges, including $(+0, -0)$ in the case of the TSPPD. If only $|S| - 1$ edges from a subtour can be in the final tour, there must be $n - (|S| - 1) = n - |S| + 1$ edges that are not part of the subtour in the final solution.

Pferschy and Staněk (2016) [36] investigate the benefits of switching between constraints (9) and (3), observing that the solution time for a MIP tends to increase with the density of its constraint matrix. They choose between the two formulations based on the number of nodes in each subtour as in the form (10).

$$\begin{array}{l} \sum\limits_{i,j \in S} x_{ij} \leq |S| - 1 \text{ if } |S| \leq \frac{n+1}{3} \\ \quad x\left(\delta\left(S\right)\right) \geq 2 \text{ if } |S| > \frac{n+1}{3} \end{array} \tag{10}$$

We integrate SEC into the TSPPD by first eliminating subtours in Gurobi's callback. Once there is only one tour, that is, $|\mathcal{T}| = 1$, the current solution is a valid TSP solution. We then scan it for precedence violations as described in algorithm 2, adding lazy cuts using constraint (4) as precedence violations are found.

Algorithm 2 TSPPD MIP Precedence Callback

```
tour ← an ordered vector starting at +0 and ending at −0
S ← ∅
for all i ∈ {1, ..., n} do
    current ← tour(i)
    S ← S ∪ {current}
    if current ∈ V_ and pickup(current) ∉ S then
        Σ_{i∈S, j∈V\S} x_ij ≥ 4
    end if
end for
```

Figure 2 shows the median log time to prove optimality for TSP and TSPPD instances of different sizes based on the SEC forms described in Pferschy and Staněk (2016) [36]. For each problem size we generate 25 instances with random node locations drawn uniformly from a square region. The subtour form uses constraint (9), while the cutset form uses constraint (3). The hybrid form switches between these forms as in constraint (10).

Pferschy and Staněk (2016) [36] use the SCIP optimization solver, adding SEC at the end of the optimization and re-optimizing if subtours are found [29]. Our implementation uses lazy constraints in Gurobi [21]. We observe some of the same effects in TSPs that they do, namely the subtour form dominates the cutset form in performance, and the hybrid form is most effective in larger instances, particularly those over 50 nodes. We don't observe much benefit to using hybrid SEC for the TSPPD, and therefore opt to use the subtour form shown in constraint set (9) for the remainder of the paper.

Our version of the MIP model for solving the TSPPD thus starts by finding infeasible solutions and works toward feasibility through the addition of SEC and precedence constraints. While this can be very effective for optimization, it does mean that feasibility may not be realized early in the search without some external modification of the solution. As with other MIP exact solvers, we warm start the search with solutions from a CP search.

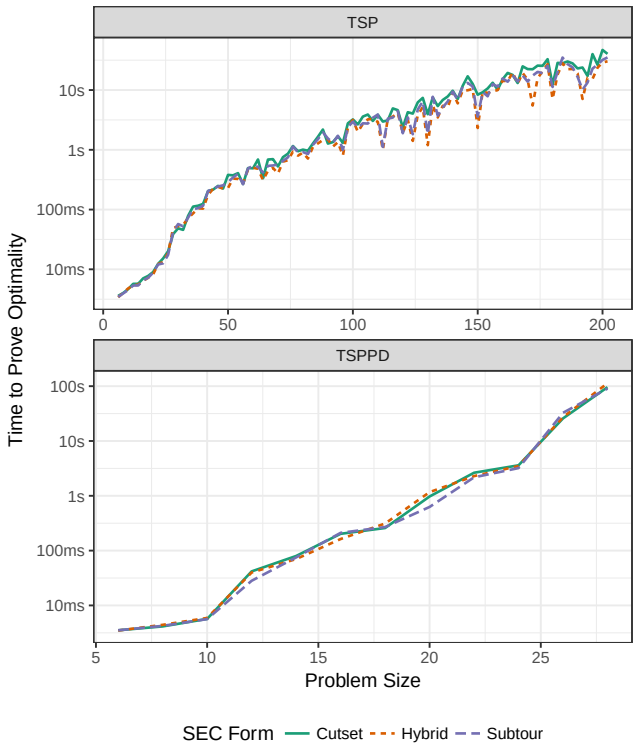

Fig. 2 Median time to prove optimality of SEC forms.

2.4 Polynomial Length ATSPPD Models

Not all TSP formulations require relaxation. A number of models contain constraints that imply the Dantzig-Fulkerson-Johnson subtour elimination constraints. Perhaps most notably, the Miller-Tucker-Zemlin formulation use variables to indicate which position each node is in a feasible tour [31].

Sarin et al. (2005) [39] introduce a polynomial-length ATSP model based on an assignment problem formulation. Their model uses x variables to represent directed arcs that indicate direct precedence and y variables to indicate route precedence. If $x_{ij} = 1$ then i directly precedes j, and if $y_{ij} = 1$ then i precedes j but need not directly precede it. This models is given in Formulation 3.

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

$$\text{s.t.} \sum_{(i,j)\in A} x_{ij} = 1 \qquad \forall\, i \in V$$

$$\sum_{(i,j)\in A} x_{ij} = 1 \qquad \forall\, j \in V$$

$$x_{ij} \le y_{ij} \qquad \forall\, i,j \in V \qquad (11)$$

$$y_{ij} + y_{ji} = 1 \qquad \forall\, i,j \in V, i \neq j \qquad (12)$$

$$y_{ij} + x_{ji} + y_{jk} + y_{ki} \le 2 \qquad \forall\, i,j,k \in V, i \neq j \neq k \qquad (13)$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i,j \in V, i \neq j$$

$$0 \le y_{ij} \le 1 \qquad \forall\, i,j \in V, i \neq j$$

Formulation 3: The ATSP Formulation of Sarin et al. [39]

Constraints (11) establish the relationship between route and direct precedences. Constraints (12) require that, for each pair of nodes, one must precede the other. Subtour elimination is handled through route precedence using constraints (13). The ATSP model of Sarin et al. (2005) [39] conveniently adapts to the ATSPPD by setting $y_{+0,i} = y_{i,-0} = 1$ for all $i \in V \setminus \{+0, -0\}$, and all $y_{+i,-i} = 1$, which we present results for in section 4.

2.5 Constraint Programming Methods

A common CP formulation of the TSP employs a vector of successors as its primary decision variable [9, 17, 35]. Given nodes V, `next` is a vector of length $n = |V|$ where each position is associated with a single node and is assigned the value of the node that comes directly after it in the tour.

$\texttt{next} \in V^n$

Caseau and Laburthe (1997) [9] constrain the model such that no node can succeed itself, each node has a unique successor, and there are no cycles in any feasible solution. In their version, the `alldifferent` global constraint is implemented using a standard library. They enforce `nocycle` in Formulation 4 by failing solutions that contain multiple subtours throughout the search tree [9, 24].

$\texttt{next}_i \neq i \quad \forall \quad i \in V$
`alldifferent`(`next`)
`nocycle`(`next`)

Formulation 4: TSP Feasible Space

Some modern constraint solvers such as Gecode provide a `circuit` constraint, which efficiently handles propagation for all of these and guarantees that a vector forms a Hamiltonian circuit in any feasible solution [19, 24]. Using this constraint simplifies the model.

$\texttt{circuit}(c, \texttt{next}, z)$

The `circuit` constraint accepts a variable into which it propagates the total circuit length, along with a matrix of arc costs [19]. Therefore the feasible set and objective function of the TSP can be fully represented using the fairly simple CP Formulation 5.

$$\begin{aligned} \min\ & z \\ \text{s.t.}\ & \texttt{circuit}(c, \texttt{next}, z) \\ & \texttt{next} \in V^n \end{aligned}$$

Formulation 5: Circuit-Based TSP Formulation

Formulation 5 is advantageous in that it fully describes the TSP for n nodes. No additional constraints related to subtour elimination need to be generated during search. At each step in the search tree, constraint propagation maintains a domain relaxation of the `next` vector that has not been proven infeasible. If any element of that vector is reduced to an empty domain, the solver fails that section of the search tree and continues its search elsewhere.

When it comes to the pickup and delivery portion of the TSPPD, there is less clarity in the modeling approach. Well studied constraints exist in the CP literature for implementing value precedence on vectors [25]. However, these constraints do not operate on successor vectors such as `next`.

Implementations of precedence on circuits are described in general terms in the CP literature in relation to the TSP with time windows (TSPTW), and rely on the computation of times for node arrivals and comparison of those arrival or execution times to feasible windows [5]. These formulations enforce precedence among pickup and delivery pairs using the cost of the tour upon visiting each node. A delivery's cost must be at least that of its associated pickup plus the cost of traveling along their connecting arc. We refer to this as "cost-based precedence".

Such a formulation is intended to operate on TSPs where there are time windows during which events must occur. This concept is not found in the pure TSPPD, in which precedence may be more directly modeled on circuits using set-based variables and constraints.

The most complete description we have found of a precedence constraint which solely encodes the precedence relations in a circuit is described in Focacci et al. (2002) [16]. Their formulation maintains predecessor and successor sets for each node. They use this mechanism to detect when additional nodes must be inserted between node pairs, and to assist in domain reduction on the `next` vector. We adapt and formalize this representation for the TSPPD in section 3.2 and compare it to the cost-based precedence used for the TSPTW.

2.5.1 Branching and Propagation

A CP solver's search over a model's feasible space is guided by branching on variable domains. At each branch point, the domains of a variable or set of variables are reduced. The constraints of the model propagate these changes into the domains of other variables in the model that are either directly or indirectly related to the branching variables. When a variable's domain is reduced to a single value, it is assigned that value for any nodes further down the search tree. If propagation reduces a variable's domain to the empty set, that segment of the search tree fails and is removed from consideration.

Unlike MIP solvers that obtain lower bounds from the LP relaxation, CP solvers traditionally do not naturally include a mechanism for obtaining relaxation bounds for the problem. Instead, VP uses propositional logic and clever search techniques to fathom major parts of the tree and is often stopped as soon as a feasible solution is found. Thus, it is more often used as a satisfiability algorithm than as an optimization procedure. There are some papers that explore the integration of reduced cost and dual bound computations into CP search [16, 17, 30]. We present test results for such formulations in section 4.

It is well established that the `path` or `circuit` constraints of CP solvers are not effective at finding optimal TSP solutions on their own [17]. In order to aid the optimization process, we direct the search through branching rules and additional propagation based on objective value bounds. As has long been understood with MIP, dynamic variable selection is fundamental to fast solution of problems in CP, particularly when such problems require optimization of some objective function [35].

Often, CP modelers adopt heuristic techniques to determine variable branching order and choose the most profitable assignments. One such variable selection technique is the `first-fail` heuristic, in which the variable with the smallest domain at the current point in the search tree is branched on, generating early failures in an attempt to fathom sections of the search tree quickly. Both Caseau and Laburthe (1997) [9] and Pesant et al. (1998) [35] use `first-fail` as a component in their

branching schemes to select among the unassigned `next` variables.

Pesant et al. (1998) [35] use a branching strategy that adds a tie-breaking rule to `first-fail`. If two variables have the same, minimal, domain cardinality, they select the one that appears in more domains of other unassigned variables. They then assign successor nodes to the branching variable in increasing order of their cost, measured by the arc cost incurred by fixing the connection between the branching variable and its assigned successor.

Caseau and Laburthe (1997) [9] offer a more sophisticated branching scheme, which incorporates a `regret` heuristic. At each node in the search tree, the regret for any unassigned node i in the `next` vector is the difference between its two closest, or lowest cost, feasible assignments, j_1 and j_2, as shown in equation (14).

$$\text{regret}(i) = |c_{ij_1} - c_{ij_2}| \tag{14}$$

Branching on variable regret can help a CP solver make good choices early in the search. Consistent with this idea is the view of regret as an approximation of reduced costs of an assignment problem (AP) relaxation of the TSP [17]. A weakness of the regret heuristic is that it is not as straightforward in adapting to side constraints as some others, such as closest neighbor (CN), which implements a pure greedy strategy. CN branching is easily modified to respect innately precedence rules by starting at +0 and always branching on the least cost feasible arc at the end of the current partial route. We call this branching scheme Sequential Closest Neighbor (SCN). Regret branching is not so easily adapted, in our experience, and requires effective propagation using decisions made about partial routes.

2.5.2 Bounding Techniques

A particular difficulty in using CP techniques for optimization is the handling of objective functions. A standard approach is to assign a variable to the objective value. When a new incumbent solution is found, we require that all subsequent solutions be better than our best known solution. Meanwhile, as other variable domains are restricted as a result of choices in the search tree, lower bounds are set on the objective variable. This allows the solver to fathom sections of the search tree that cannot perform as well as the current incumbent solution [17]. In a CP system such as Gecode, this requires adding a constraint on the cost $z < z^*$ when a new incumbent solution with cost z^* is discovered in the search tree.

Caseau and Laburthe (1997) [9] and Pesant et al. (1998) [35] explore three different mechanisms for establishing lower bounds on TSP cost at each branch in the search tree. CN bounds are computed by summing the cost of assigning each node to the closest node in its domain. These bounds are easy to compute, but myopic and tend not to be tight. Such bounds are easily computed as part of variable domain propagation in the `circuit` constraint, and thus do not require an explicit representation.

More sophisticated bounds involve computing a Minimum Spanning Tree (MST) over the domain of the `next` vector. A MST does not require that its solution form a tour and thus underestimates the cost of the TSP. Computing MSTs is fast and it is possible to quickly update the solutions within the search tree using an incremental form of Kruskal's algorithm [9, 35].

Alternatively, one can derive bounds from a Minimum Spanning Arborescence (MSA) over the domain of the `next` vector. A MSA forms a directed spanning tree from the root towards its leaves. MSAs can be computed in $O(EV)$ time using Edmonds' algorithm [13]. More recent refinements of the algorithm can also be deployed that find an MSA in $O(E + V \log V)$ time [18].

Focacci et al. (1999) [17] and Focacci et al. (2002) [16] explore the idea of using relaxations during the search that provide both dual bounds and reduced cost information for variable fixing and triggering further propagation. They implement an Assignment Problem (AP) relaxation of the TSP using the feasible domain of `next` in the search tree, and another vector `prev`, which is the inverse of `next`. Only feasible arcs at the current location in the search tree are included in the AP relaxation.

The AP relaxation can be solved in $O(n^3)$ time at the root node in the search tree, and in $O(n^2)$ time upon domain changes throughout the tree using augmenting path steps [8]. A reduced cost matrix is computed as part of these updates, and can be then used to prune variable domains from `next`. Empirical results show this technique is more efficient with respect to solution time than MST-based dual bounds due to its relatively low computational complexity [16, 17].

2.5.3 Additive Bounding & 1-Tree Relaxations

Benchimol et al. (2012) [4] explore techniques to increase filtering of weighted circuit constraints by combining AP relaxations with 1-tree relaxations obtained from Held-Karp bounds [22, 23]. They combine these two relaxations within the additive bounding framework of Fischetti and Toth (1992) [14]. This procedure

uses one relaxation to compute an initial dual bound and set of reduced costs, then applies subsequent relaxations of different forms directly to the reduced cost matrix resulting from the initial relaxation. The final dual bound is the sum of the objective values of the various relaxations. The reduced cost matrix is provided by the final relaxation applied [14].

Additive bounding can provide tighter dual bounds than individual relaxations because each relaxation incorporates different constraints from the original model. For instance, an AP relaxation has degree constraints but will allow subtours, while a 1-tree relaxation disallows subtours while not enforcing degree constraints. Combining the relaxations can enhance filtering inside a CP search and may result in dramatically smaller search trees, as shown in Benchimol et al. (2012) [4].

We attempt to improve the runtime of our CP models against TSPPD instances using a similar technique. However, while incremental AP relaxations fit easily into real-time optimization since AP updates require a single execution of an $O(n^2)$ algorithm with each relevant variable domain change, adapting 1-tree relaxations to the demands of real-time systems is not so straightforward. Computing an optimal 1-tree relaxation requires repeated applications of an algorithm such as Kruskal's, each of which runs in $O(m \log n)$ time [4, 40].

Detecting convergence of the 1-tree bound is nontrivial and based on subgradient techniques [15]. Each iteration involves calculating *node potentials* which penalize nodes that do not have a degree of two, as in (15). t^m is the current step size, while d_i^m is the degree of node i in iteration m. The node potentials π are added into the edge costs for the next 1-tree as $c_{ij} + \pi_i + \pi_j$, where c_{ij} is the original edge cost. Since a TSP tour incurs each node potential cost twice, the TSP is invariant to these node potentials, while the 1-tree is not [23]. Converting a 1-tree bound into a TSP bound simply involves subtracting each node potential twice from the 1-tree cost.

$$\begin{aligned} \pi_i^1 &= 0 && \forall\ 1 \le i \le n \\ \pi_i^{m+1} &= \pi_i^m + t^m(d_i^m - 2) && \forall\ 1 \le i \le n \end{aligned} \tag{15}$$

A number of mechanisms exist for updating the step size t^m. We compute 1-trees using a fixed number of iterations, as outlined in Valenzuela and Jones (1997) [40]. The formula (16) gives a step size update that works on a fixed number of iterations, where L^1 is the first 1-tree objective value, m is the current iteration, M is the total number of iterations, and n is the number of nodes in the graph.

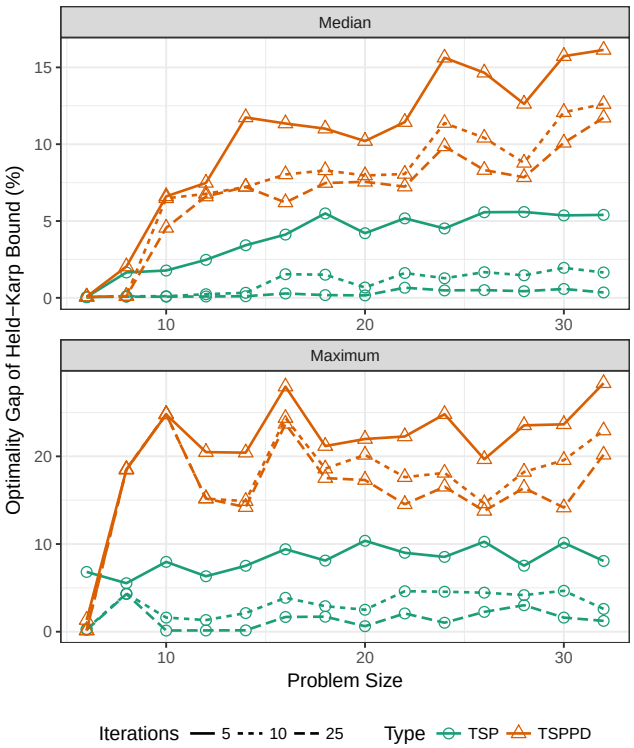

Fig. 3 Optimality gap of Held-Karp bounds.

$$\begin{aligned} t^1 &= \tfrac{1}{2n} L^1 \\ t^m &= t^1 \left((m-1)\tfrac{2M-5}{2(M-1)} - (m-2) + \tfrac{(m-1)(m-2)}{2(M-1)(M-2)} \right) \end{aligned} \tag{16}$$

The Held-Karp bound at a node in the CP search is thus the maximum objective in terms of the original edge costs from a series of M 1-tree computations. Variable domain filtering can be accomplished using either marginal costs per Benchimol et al. (2010) [3] or by computing reduced costs directly. While we observe both improvements in runtime and search tree size for TSP instances using the 1-tree relaxation and additive bounding, those benefits do not translate readily to TSPPD instances in our implementation. This appears to be because the 1-tree bounds degrade significantly in the presence of precedence constraints.

Figure 3 shows median and maximum optimality gaps of the Held-Karp bound after a fixed number of iterations against TSP and TSPPD instances by number of nodes. We use the Grubhub test instances, described in section 4. The TSP instances are simply the TSPPD instances with the precedence constraints relaxed. Optimality gap is computed as $(z^* - w^*)/w^*$ where z^* is the true optimal solution to a TSP or TSPPD instance, respectively, and w^* is the 1-tree objective value in terms

of the original edge costs at the root node of the search tree after a fixed number of iterations.

We observe that, as problem size grows, the Held-Karp bound remains quite good for TSP instances, while degrading quickly for TSPPD instances. In the update mechanism we use, the benefit of increasing the number of iterations drops sharply from 25 to 100. Further, the spread of the optimality gap appears to decrease for the TSP with increased iterations, but not for the TSPPD. We believe that this challenge is due to the structure of the Held-Karp bound when applied to TSPPD instances. The bound itself is based on undirected graphs, while the TSPPD is fundamentally a directed problem (pickups must precede deliveries). Results against TSP instances indicate that adapting the Held-Karp bound to the TSPPD may be a worthy endeavor, but we do not consider it further in this paper.

3 Approach

Our goal in this study is not to directly compare MIP and CP methods. While the latter can outperform MIP for certain problem classes, MIP solvers have been traditionally designed and built expressly for optimization. Rather, we hope to observe the impact of pickup and delivery structure applied to small TSPPD instances which must be solved in "real time", when high quality routes must be constructed within seconds or milliseconds in response to events during an operational period. Intuitively, we understand that additional constraints complicate the search for MIP solvers, while supporting it for CP. It is less clear how precedence impacts solution time, and whether those solution times can be improved through hybridization, i.e. using components of more than one standard algorithm.

3.1 Mixed Integer Programming TSPPD Models

We base our MIP implementation of the TSPPD solver on that of Pferschy and Staněk (2016). There are a few points where our underlying TSP implementation diverges from theirs. First, they use the SCIP solver as their optimization engine, while we use Gurobi [21, 29]. They mention the possibility of using lazy constraints for SEC but do not actually implement and test this approach. We implement subtour elimination inside lazy constraints using inequality (9) in our implementation and precedence constraints using inequality (4).

Once a single tour is found in the search tree, we scan it for precedence constraint violations. If a partial route S exists such that $+0, -i \in S$ and $-0, +i \notin S$, we add a lazy constraint using inequality (4). This results in model (1).

Algorithm 2 detects precedence violations given feasible tours by starting at $+0$ and following the directed tour from that node. If it finds a delivery that occurs before its associated pickup, it adds a lazy constraint. For comparison, we also test the asymmetric polynomial-length model in Formulation 3 with precedence of pickup and delivery pairs enforced by setting $y_{+i,-i} = 1$ for each pair $(+i, -i)$.

3.2 Constraint Programming TSPPD Model

3.2.1 Cost-Based Circuit Precedence

The addition of pickup and delivery precedence relations to the CP TSP model requires a constraint that fails the search tree when they are violated and removes infeasible values from variable domains in response to branching decisions. Berbeglia et al. (2012) [5] give a formulation that enforces precedence using the times that events occur, a common structure in TSPTW models.

Since the pure form of the TSPPD does not include time windows, we think of this as cost-based precedence, as formulated in (17). A `cost` vector encodes the cumulative route cost at every node. Each delivery $-i \in V_-$ must have a `cost` greater than or equal to its associated pickup $+i$.

$$
\begin{aligned}
&\min\ z \\
&\text{s.t.}\ \ \texttt{circuit}(c, \texttt{next}, z) \\
&\quad \texttt{circuit-precede-cost}(\texttt{next}, \texttt{cost}) \\
&\quad \texttt{next}_{-0} = +0 \\
&\quad \texttt{cost}_{+0} = 0 \\
&\quad \texttt{cost}_{-i} \geq \texttt{cost}_{+i} \qquad\qquad \forall\, +i \in V_+ \\
&\quad \texttt{next} \in V^n \\
&\quad \texttt{cost} \in \mathbb{N}^n \\
&\quad z \geq \max \texttt{cost}
\end{aligned}
\tag{17}
$$

A custom propagator `circuit-precede-cost` detects when a branching decision is made such that $\texttt{next}_i = j$ in the search tree, either directly through branching or as a result of propagation. The assignment of $\texttt{next}_{-0}$ is required by the model and ignored during propagation. When any other variable $\texttt{next}_i$ is assigned a value j in the search tree, the `cost` i must now be equal to the `cost` of j plus the cost of their connecting arc. This is achieved through a local constraint of the form (18).

$$
\texttt{next}_i = j \implies \texttt{cost}_j = \texttt{cost}_i + c_{ij} \tag{18}
$$

3.2.2 Set-Based Circuit Precedence

It is also possible to enforce precedence more directly using set variables, as Focacci et al. (2002) [16] outline. Our implementation uses vectors of predecessor and successor nodes called `pred` and `succ`, respectively, that track which nodes must and must not precede and succeed each node based on the current structure of partial routes in the solution. The TSPPD CP formulation using set precedence is shown in Formulation 19.

A custom propagator `circuit-precede-set` is triggered whenever an assignment to `next` is made. The origin node, $+0$, must have an empty set of predecessors, while the destination node, -0, must have an empty set of successors. No node's predecessor and successor sets are allowed to intersect. A pickup's successor set must be a superset of its associated delivery's successor set, and the delivery's predecessor set must be a superset of the pickup's predecessor set.

$$
\begin{aligned}
&\min\ z \\
&\text{s.t.}\ \ \texttt{circuit}(\texttt{next}, z) \\
&\quad \texttt{circuit-precede-set}(\texttt{next}, \texttt{pred}, \texttt{succ}) \\
&\quad \texttt{next}_{-0} = +0 \\
&\quad +i \in \texttt{pred}_{-i} && \forall\ -i \in V_D \\
&\quad -i \in \texttt{succ}_{+i} && \forall\ +i \in V_P \\
&\quad \texttt{pred}_i \cap \texttt{succ}_i = \emptyset && \forall\ i \in V \\
&\quad \texttt{pred}_{-i} \subset \texttt{pred}_{+i} && \forall\ +i \in V_+ \\
&\quad \texttt{succ}_{+i} \subset \texttt{succ}_{-i} && \forall\ +i \in V_+ \\
&\quad \texttt{pred}_{+0} = \emptyset \\
&\quad \texttt{succ}_{-0} = \emptyset \\
&\quad \texttt{next} \in V^n \\
&\quad \texttt{pred} \in \{V^n\}^n \\
&\quad \texttt{succ} \in \{V^n\}^n \\
&\quad z \geq 0
\end{aligned}
\tag{19}
$$

The assignment of $\texttt{next}_{-0}$ is required by the model and ignored during propagation. When any other variable $\texttt{next}_i$ is assigned a value j in the search tree, the successor set for i must now be equal to the successor set of j with the addition of j itself. Similarly, the predecessor set of j must now be equal to the predecessor set of i with the addition of i. Our `circuit-precede-set` constraint shown in (20) accomplishes this by adding local constraints to the current branch of the search tree upon assignment of any $\texttt{next}_i$.

$$
\begin{aligned}
\texttt{next}_i = j \implies\ & \texttt{pred}_j = \texttt{pred}_i \cup \{i\} \\
& \texttt{succ}_i = \texttt{succ}_j \cup \{j\}
\end{aligned}
\tag{20}
$$

$$
\begin{aligned}
\texttt{next}_{+i} = +j \implies\ & -i \in \texttt{succ}_{+j} \\
& -j \in \texttt{succ}_{+i} \\
\\
\texttt{next}_{+i} = -j \implies\ & -i \in \texttt{succ}_{+j} \\
& -i \in \texttt{succ}_{-j} \\
& +j \in \texttt{pred}_{+i} \\
& +j \in \texttt{pred}_{-i} \\
\\
\texttt{next}_{-i} = +j \implies\ & +i \in \texttt{pred}_{+j} \\
& +i \in \texttt{pred}_{-j} \\
& -j \in \texttt{succ}_{+i} \\
& -j \in \texttt{succ}_{-i} \\
\\
\texttt{next}_{-i} = -j \implies\ & +i \in \texttt{pred}_{-j} \\
& +j \in \texttt{pred}_{-i}
\end{aligned}
\tag{21}
$$

Further propagation is accomplished by explicitly adding to and removing values from the predecessor and successor sets upon propagation of $\texttt{next}_i = j$, depending on the values of i and j and what is known about their precedence relations, as shown in (21). This allows failures to be detected early in the search tree.

Set-based precedence operates directly on precedence structure and does not require time windows. However, it may be beneficial to combine both forms in cases where there is precedence structure and time windows on event times. We study a final form of precedence constraint that combines the cost and set-based form by simply applying both of them.

3.3 Hybrid TSPPD Models

Milano et al. (2001) [30] characterize the benefits of incorporating optimization procedures and CP into the same framework. We attempt two hybrid methods for tackling the TSPPD. In the first, we incorporate AP relaxations for reduced cost variable fixing into our CP TSPPD model, similar to its application to the TSPTW by Focacci et al. (2002) [16]. In the second, we attempt to combat the issue of MIP starting its search far from feasibility by using a CP model to find solutions quickly, and warm-starting the MIP solver with the best known solution after a given time limit.

3.3.1 Reduced Cost-Based Domain Filtering

We implement reduced cost-based variable fixing as a custom propagator inside of Gecode CP framework. The propagator subscribes to value assignments on each index of the `next` vector. When first triggered, the propagator initializes and solves the primal-dual AP solving

algorithm of Carpaneto et al. (1988) [8], an operation that takes $O(n^3)$ time. When subsequent assignments to `next` occur in the search tree, the propagator performs an augmenting path update in $O(n^2)$ time.

The custom propagator tracks which variables are currently unassigned. When a branching decision is made such that $\texttt{next}_i = j$, the propagator forces the primal-dual algorithm to set row i to column j in its internal matrix, and removes any conflicting assignments from consideration. For each unassigned variable $\texttt{next}_i$ and each j in the domain of $\texttt{next}_i$, the AP relaxation provides a reduced cost rc_{ij}. It also provides a dual bound w^* for the objective function of the TSPPD at that location in the search tree. If, for any feasible arc (i, j), $w^* + rc_{ij} \geq \hat{z}$, where $\hat{z}$ is the upper bound on the objective function, we infer that $\texttt{next}_i \neq j$.

3.3.2 MIP Warm Starting

We study a second form of integration by using CP to warm start MIP. As previously discussed in section 1.2, side constraints can make MIP implementations take longer to find feasible solutions. We address this behavior by using our CP implementation to find good solutions quickly, and then warm starting the MIP solver with them. This not only provides primal bounds to the MIP solver, but also gives it a neighborhood in which to search. We test the warm starting technique using different time budgets on the CP solver.

4 Results

We test enumerative, MIP, CP, and hybrid forms of the TSPPD models using instances constructed from pickup and delivery locations observed at Grubhub, along with expected travel times connecting location pairs. The test set has 10 instances per problem size. This allows us to gauge the performance of the models on realistic problems from meal delivery, in which pickup locations are likely to be clustered close together, and there may be many near optimal solutions that would have to be considered to prove optimality.

We evaluate the models according to multiple criteria: the time taken to find a first feasible TSPPD route, the time to find a route within 10% of the true optimum, the time to find an optimal route, and the time to prove optimality. The first three measures are important to real-time logistics, where high quality routes must be found quickly and there may not be enough time to optimize fully. The final measure quantifies the capacity of model formulations and algorithms to optimize. We use time to prove optimality as a proxy for general algorithmic performance while selecting configurations. Not all modeling techniques surveyed have access to global dual bounds, so we measure optimality gap of primal solutions from the true optimum.

Charts report the median and maximum of execution times for each problem size and model configuration with the vertical axis scaled to log time. These measures are useful in choosing models for production systems because they give us a sense of typical performance and poor performance in terms of execution time. Given a target problem size and time budget, we can use the results of these test to determine the best approach for a given TSPPD problem size. Execution times are limited to 1000 seconds. Any model configuration that is not able to achieve a particular goal (e.g. finding a feasible solution, or finding a solution with 10% of optimal) for all instances of a given size is removed from consideration for that problem size, causing its line to end on the graph.

We generate results using Gurobi 8.0.0 as the MIP solver and Gecode 6.0.1 for the CP models. Model code is written using C++14. The test machine is a Lenovo X1 Carbon with a 4-core Intel Core i5 CPU and 16 GB of RAM. We test using 1 thread and 4 threads, but concurrency does not appear to change our conclusions, so we report results for a single thread. Single-threaded executions of these models are deterministic and thus easier to interpret and compare.

Sections 4.1 and 4.2 discuss the testing of the CP alternatives. Section 4.3 then similarly discusses our testing of MIP models. Finally, section 4.4 provides results that compare enumeration to the best CP and best MIP models.

4.1 Circuit Precedence Forms

Figure 4 shows the performance characteristics of cost-based precedence (Formulation 17), set-based precedence (Formulation 19), and the combination of those two model formulations. We observe the best and most consistent performance from set-based precedence with either regret or SCN branching. Combined precedence performs similarly to set-based, indicating that there is a benefit to adding set-based precedence to circuits that involve time windows, when applicable. Due to its performance, we fix the CP TSPPD models to use set-based precedence for the remainder of the paper.

4.2 Reduced Cost-Based Domain Filtering

Figures 5 and 6 shows the speedup obtained by the CP TSPPD solver due to incorporating reduced cost-based

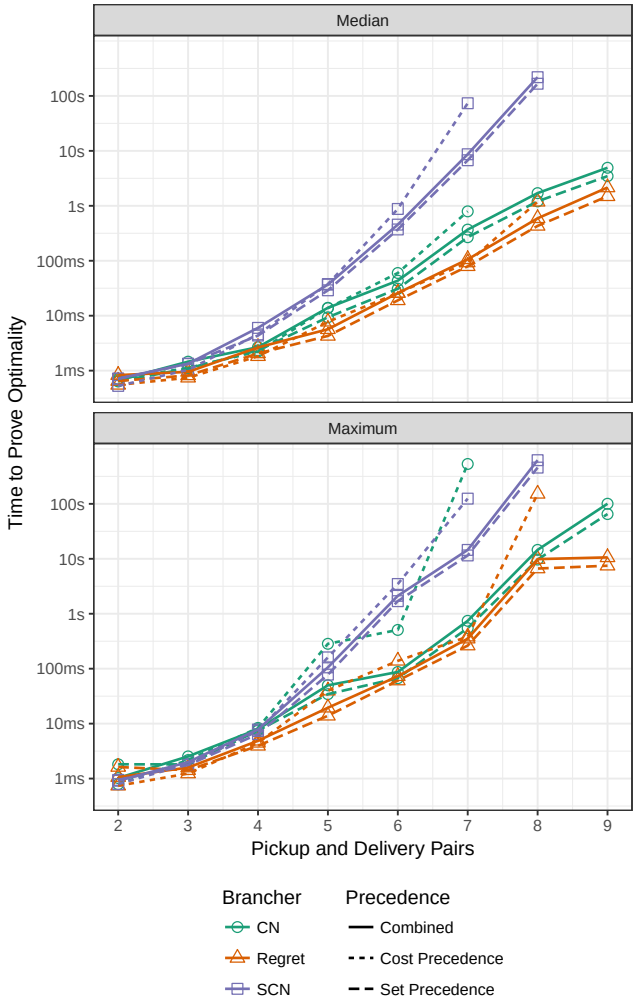

Fig. 4 Precedence forms in TSPPD CP models.

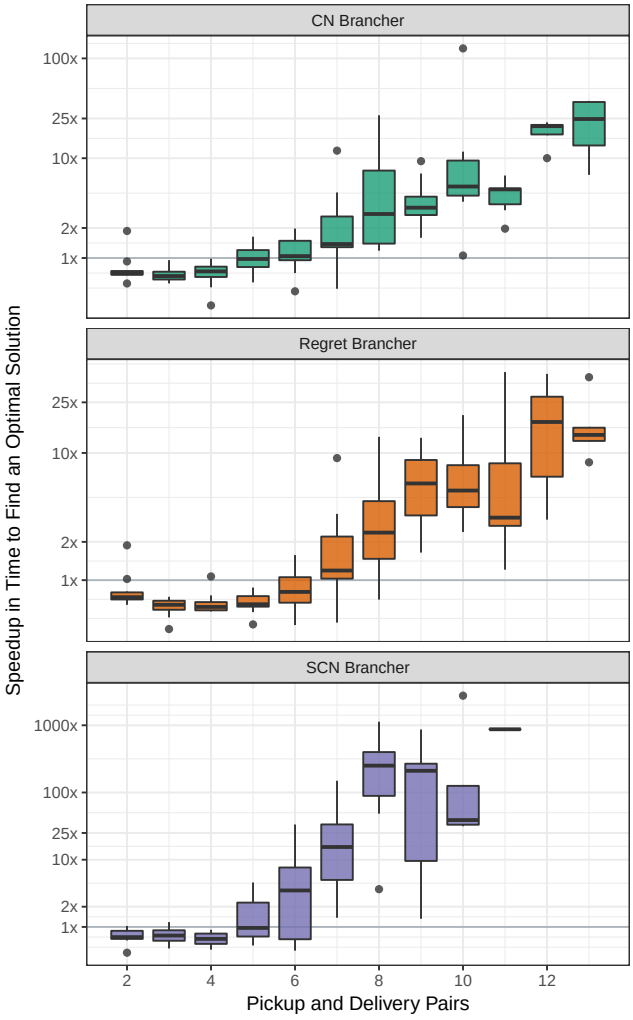

Fig. 5 Speedup finding optimal solutions due to AP relaxation and reduced cost fixing.

	Pickup and Delivery Pairs					
Brancher	2	3	4	5	6	7
All	0.74	0.72	0.90	1.20	1.57	3.98
CN	0.74	0.71	0.85	1.07	1.18	1.80
Regret	0.75	0.67	0.71	0.80	1.02	1.59
SCN	0.74	0.83	1.45	4.44	14.0	127
	8	9	10	11	12	13
All	8.35	4.89	6.69	4.11	19.4	16.2
CN	2.52	3.76	6.95	4.37	23.0	53.3
Regret	3.24	5.35	5.94	3.84	16.8	13.6
SCN	520	480				

Table 1 Median speedup to proving optimality from incorporating AP relaxation and reduced cost fixing.

variable domain filtering using the AP relaxations. The columns correspond to CN, regret, and SCN branching. Table 1 shows median speedups for proving optimality.

We observe that reduced cost-based domain filtering begins to improve the median solution time for finding optimal solutions at 6 pickup and delivery pairs, and for proving optimality at 5 pairs. The speedup acquired appears to increase with the problem size. Further, filtering brings the poor performance of SCN branching closer in line with regret branching.

Figure 7 shows that, with filtering, SCN branching actually finds high quality solutions significantly faster than regret branching, while Figure 8 shows that regret branching remains better for finding optimal solutions. Subsequent results for CP solvers in this paper will always use reduced cost-based domain filtering with either regret or SCN branching.

4.3 MIP Warm Starting

Figures 9 and 10 show median and maximum times to obtain a feasible TSPPD solution and to find an optimal solution for the MIP models. The MIP solver is warm started by the CP solver configured to use set precedence, reduced cost-based variable domain filtering, and SCN branching. Setting the warm start time time to

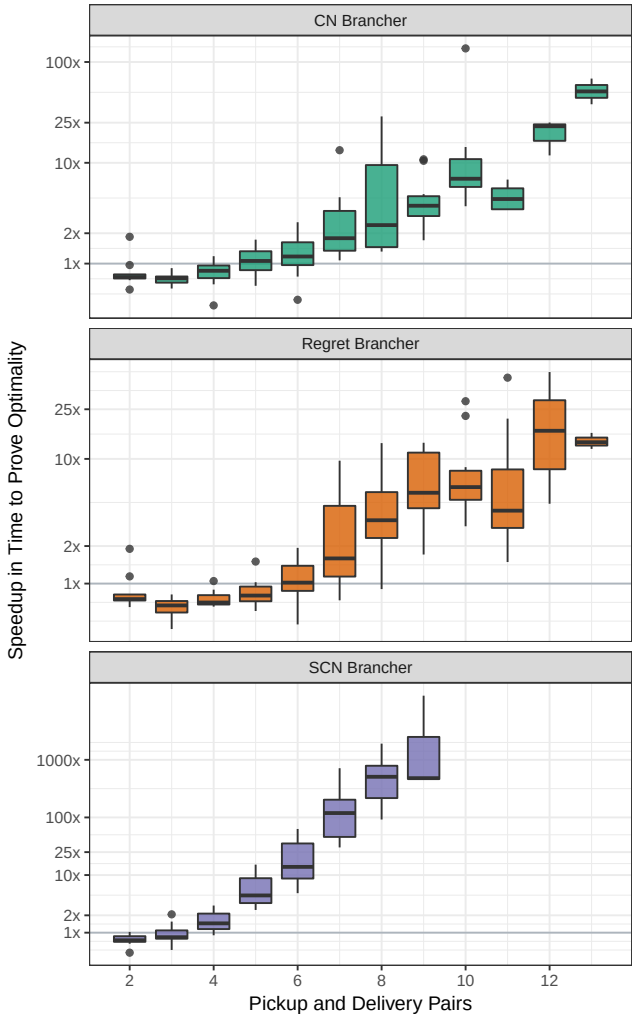

Fig. 6 Speedup proving optimality due to AP relaxation and reduced cost.

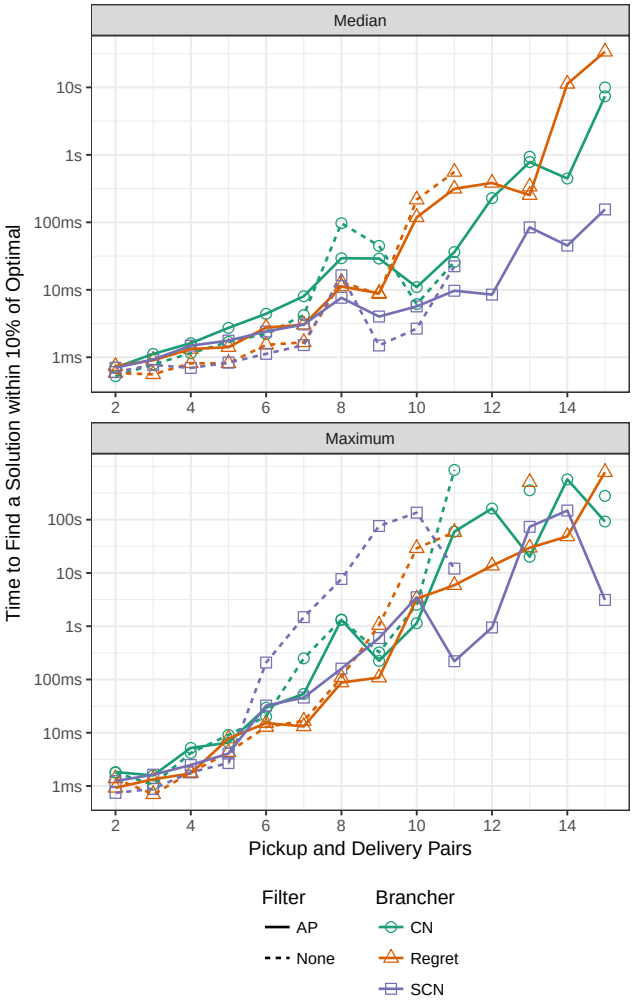

Fig. 7 Time to find a solution within 10% of optimal for TSPPD CP models with and without AP filtering.

zero milliseconds equates to using the MIP solver without a warm start.

We observe that, as problems get larger, the Ruland model takes more time to find a feasible solution. Warm starting significantly reduces its time to feasibility. Finding actionable solutions quickly is vital to real-time logistics, so it does not appear that our MIP model by itself is useful in that context for moderately sized TSPPD instances. The Sarin model, on the other hand, finds feasible solutions very quickly, but suffers from formulation overhead due to SEC and precedence constraints on the y variables. In our implementation, both CP and MIP models are constructed prior to warm starting.

The time to find an optimal solution is a measure of a method's ability to find high quality solutions quickly for real-time systems. In such systems one cares about finding good solutions, but not necessarily about proving optimality. That is, a real-time system will use the best solution it can find within its time budget, regardless of that solution's optimality gap. Time budgets for real-time path optimization vary based on the domain. In the meal delivery application, budgets of as little as 50 milliseconds are provided for the purpose of optimizing individual paths, which are often on the order of 6 to 14 nodes.

The results show a significant benefit in finding optimality for both the Ruland and Sarin models, and particularly for the latter. Presumably, this is due to the CP solver either finding an optimal solution prior to warm starting the MIP solver, or giving it a solution that is close to optimality. There does not appear to be added benefit to warm starting beyond 100 ms of CP solver time.

We observe that if we allow the MIP algorithm to run until optimality is proven, then the warm-starting does not significantly alter the total time. However,

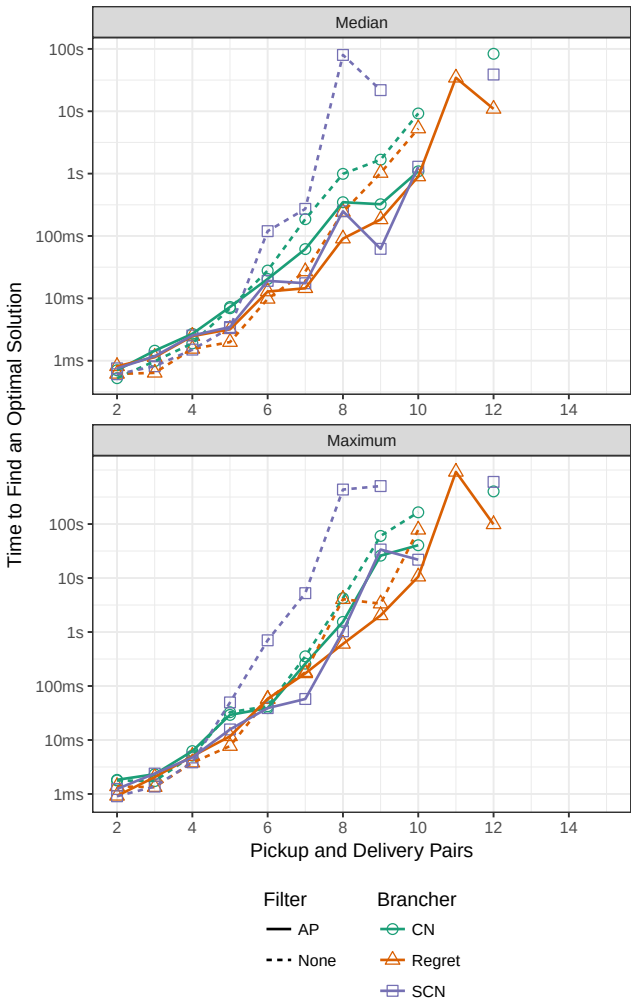

Fig. 8 Time to find an optimal solution for TSPPD CP models with and without AP filtering.

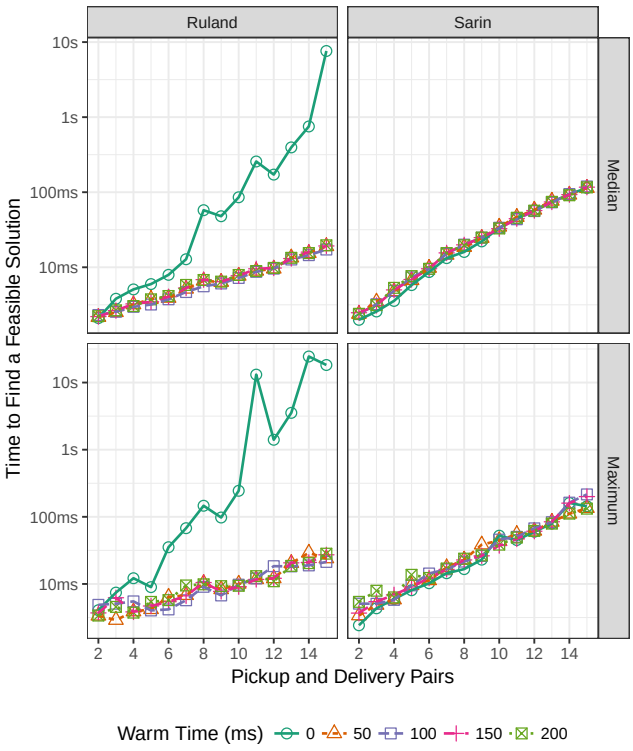

Fig. 9 Time to find a feasible solution with a Hybrid CP warm start.

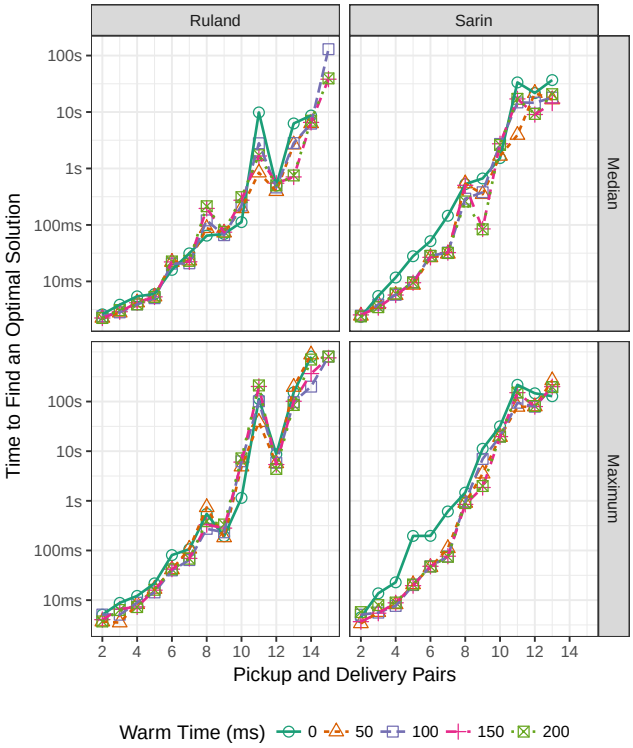

Fig. 10 Time to find an optimal solution with a Hybrid CP warm start.

when the MIP is stopped based on a time budget, the MIP procedure with the CP warm start gets to the optimal solution in reasonable times and the warm start helps it to find this solution. Since a time budget would be employed in most real-time applications, using MIP with a CP warm start works best on the larger instances.

4.4 Comparison Across Methods

In this section we show results comparing enumeration; CP with set precedence, reduced cost-based variable domain filtering, and both regret and SCN branching; MIP; and MIP with a CP warm start time of 100 ms. Charts show the median and maximum times for each problem size and event.

Figures 11, 12, 13, 14 show times to find feasible TSPPD solutions, find solutions within 10% of opti-

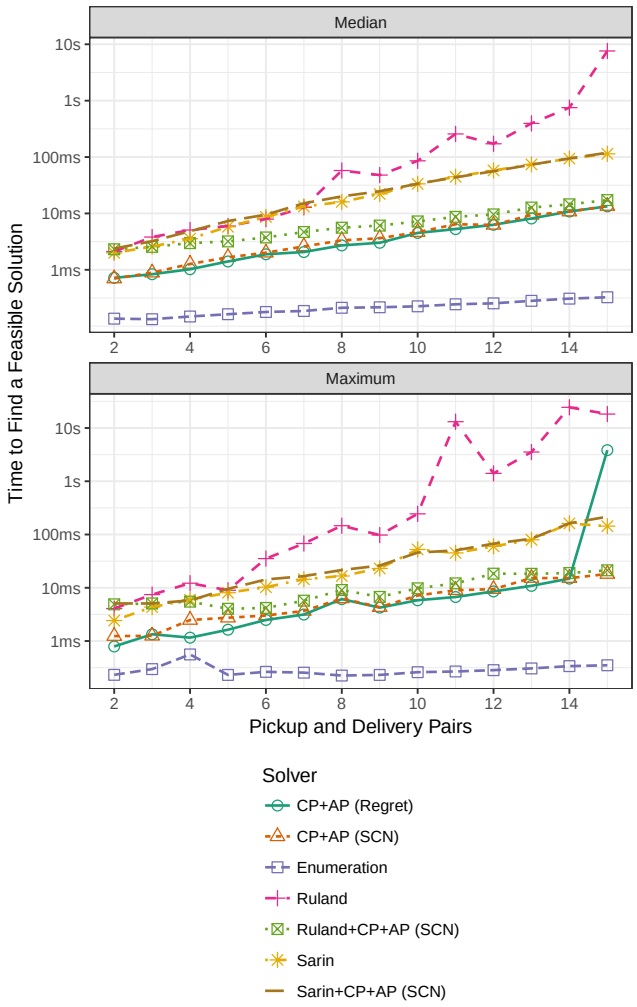

Fig. 11 Time to find a feasible solution for different solution techniques.

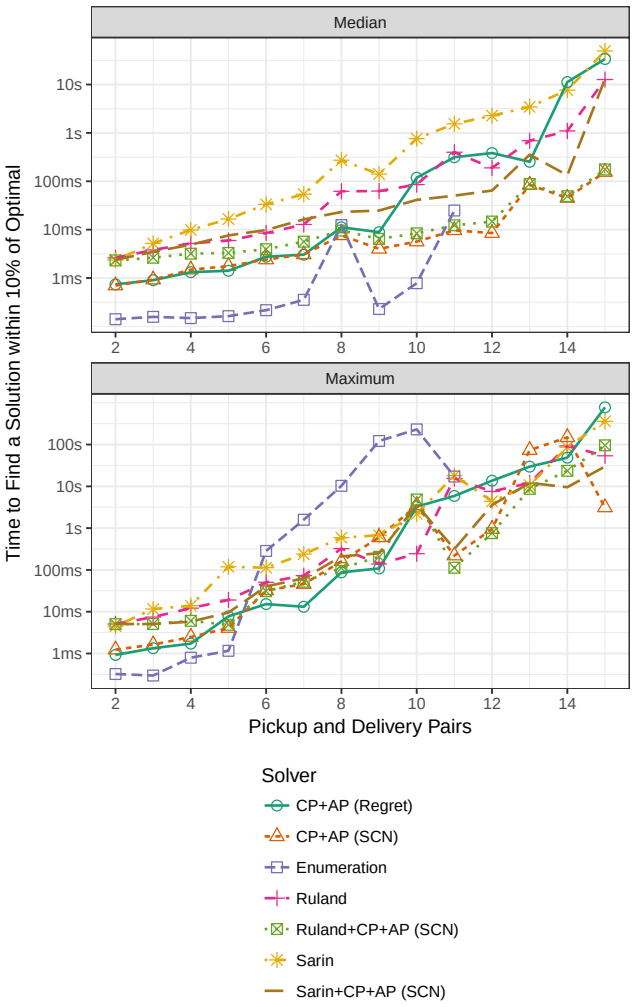

Fig. 12 Time to find a solution within 10% of optimal for different solution techniques.

mal, find optimal solutions, and prove optimality of solutions for the different solution techniques, respectively. Tables 2,3, 4, 5 provide textual versions of the median data for the Grubhub test instances.

We again observe that the Ruland MIP model requires assistance finding feasibility. CP methods tend to get feasible quickly, though they do incur some model construction overhead when compared with enumeration. The Sarin model also attains feasibility quickly during the solution process, but incurs the most overhead due to model construction. Enumeration is a good option for systems that require immediate solutions.

Enumeration is competitive at finding good solutions within 10% of optimal up to 10 pickup and delivery pairs. The Hybrid CP models with SCN branching excels at finding good solutions. Using it to warm start the Ruland model unsurprisingly yields similar performance, due to its low overhead of formulation. Warm starting the Sarin model is beneficial compared with the plain MIP model, but does not perform quite as well due to model construction.

Our enumerative implementation is competitive in finding optimal solutions for only the smallest instances, up to 5 pickup and delivery pairs, after which point its utility rapidly decreases. The graphs contain points for when a solver is able to reach a given benchmark for all instances of a certain size within 1000 seconds. Hybrid CP with regret branching outperforms pure MIP up to 7 pairs, after which point the Ruland model becomes the dominant technique. The Ruland model with a Hybrid CP warm start has similar performance, and performs best on instances with more than 10 pairs. It is the only solver able to find optimal solutions to all test instances with 15 pairs in under 1000 seconds.

Enumeration performs best in proving optimality of the smallest problems, up to 4 pairs. CP performs well

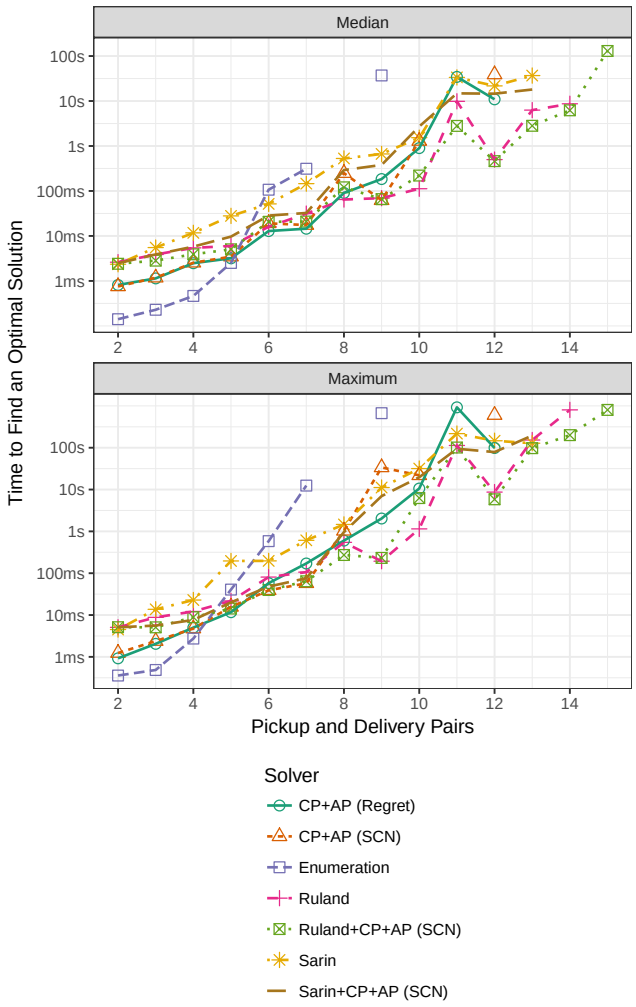

Fig. 13 Time to find an optimal solution for different solution techniques.

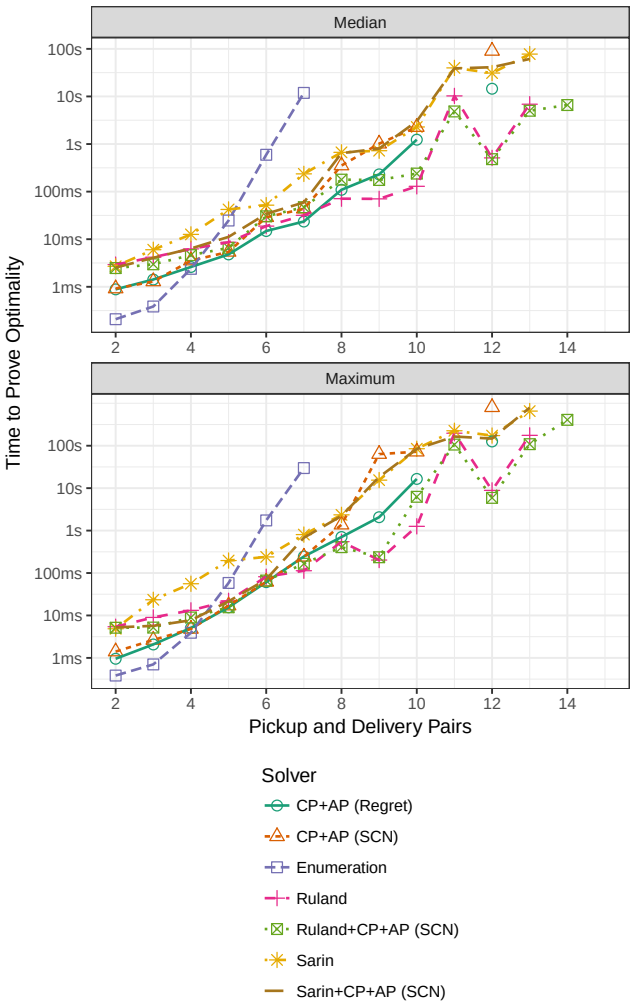

Fig. 14 Time to prove optimality for different solution techniques.

on problems up to 7 pairs. After that, the Ruland model is the dominant solution. Again, warm starting the Ruland model performs best for instances with more than 10 pairs, and the warm started Ruland model is able to fully optimize more instances within the time limit than it is without the warm start.

5 Conclusions and Future Work

Our goal is to examine the performance of various techniques for solving TSPPDs for real-time logistics, in which routes must be optimized within strict time budgets. We have implemented enumerative, CP, MIP, and hybrid approaches, and tested them using data derived from meal deliveries at Grubhub.

We find that enumeration can be effective for very small routes when implemented with intelligent arc ordering and fathoming. Enumeration scales well to obtaining feasibility in larger problem instances. Its performance degrades quickly with respect to finding good solutions and proving optimality as instance size grows.

Although CP solutions are not as fast at finding feasibility as enumeration, on average, their quality is better. Set-based precedence outperforms cost-based in our implementation, and can be used to strengthen the performance of the latter. Reduced cost-based variable domain filtering with AP relaxations is a beneficial hybridization of CP with optimization techniques, and appears to pay increasing dividends with problem size.

Hybrid CP with SCN branching is the most effective technique at finding "good" solutions (i.e. within 10% of optimal) quickly, and works well at warm starting the MIP models. Interestingly, regret branching is better at finding optimal solutions and proving optimality than SCN branching.

Solver	Pickup and Delivery Pairs 2	3	4	5	6
CP+AP (Regret)	0.7	0.8	1.0	1.4	1.9
CP+AP (SCN)	0.7	0.9	1.3	1.7	2.0
Enumeration	**0.1**	**0.1**	**0.1**	**0.2**	**0.2**
Ruland	2.1	3.8	5.1	6.0	7.9
Ruland+CP+AP	2.3	2.6	3.0	3.2	3.7
Sarin	2.0	2.6	3.6	5.8	8.6
Sarin+CP+AP	2.4	3.2	4.8	7.3	9.5
	7	8	9	10	11
CP+AP (Regret)	2.1	2.7	3.0	4.5	5.3
CP+AP (SCN)	2.6	3.3	3.6	4.6	6.4
Enumeration	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**
Ruland	12.8	57.7	47.9	85.7	257.0
Ruland+CP+AP	4.7	5.6	6.1	7.2	8.7
Sarin	13.2	16.0	22.1	33.5	44.7
Sarin+CP+AP	15.1	20.0	24.8	33.6	43.6
	12	13	14	15	
CP+AP (Regret)	6.3	8.1	10.8	13.3	
CP+AP (SCN)	6.3	9.4	10.8	13.5	
Enumeration	**0.3**	**0.3**	**0.3**	**0.3**	
Ruland	172.0	396.0	752.0	7611.0	
Ruland+CP+AP	9.6	12.7	14.5	17.2	
Sarin	57.8	74.0	93.9	114.0	
Sarin+CP+AP	56.1	73.3	94.5	119.0	

Table 2 Median milliseconds to find a feasible solution.

Solver	Pickup and Delivery Pairs 2	3	4	5	6
CP+AP (Regret)	0.8	1.1	2.5	3.2	**12.9**
CP+AP (SCN)	0.8	1.2	2.5	3.5	19.1
Enumeration	**0.1**	**0.2**	**0.5**	**2.5**	107.0
Ruland	2.6	3.9	5.4	6.0	16.0
Ruland+CP+AP	2.4	2.8	3.9	5.0	20.6
Sarin	2.3	5.5	11.7	27.9	51.6
Sarin+CP+AP	2.5	3.9	5.8	9.7	28.4
	7	8	9	10	11
CP+AP (Regret)	14.5	90.6	184.0	899	34,429
CP+AP (SCN)	**7.5**	249.0	**62.2**	1291	-
Enumeration	312.0	-	37,020.0	-	-
Ruland	31.4	**64.6**	69.3	**112**	9844
Ruland+CP+AP	20.9	120.0	65.9	220	**2798**
Sarin	146.0	528.0	666.0	1530	33,507
Sarin+CP+AP	32.4	292.0	379.0	2718	14,795
	12	13	14	15	
CP+AP (Regret)	10,911	-	-	-	
CP+AP (SCN)	38,820	-	-	-	
Enumeration	-	-	-	-	
Ruland	498	6277	8676	-	
Ruland+CP+AP	**461**	**2815**	**6220**	**129,648**	
Sarin	21,697	36,807	-	-	
Sarin+CP+AP	14,565	18,024	-	-	

Table 4 Median milliseconds to find an optimal solution.

Solver	Pickup and Delivery Pairs 2	3	4	5	6
CP+AP (Regret)	0.7	0.9	1.3	1.4	2.8
CP+AP (SCN)	0.7	0.9	1.5	1.8	2.4
Enumeration	**0.1**	**0.2**	**0.1**	**0.2**	**0.2**
Ruland	2.6	3.8	5.2	6.0	8.5
Ruland+CP+AP	2.3	2.6	3.2	3.3	3.9
Sarin	2.3	5.2	9.9	16.6	33.4
Sarin+CP+AP	2.4	3.5	4.9	7.6	9.8
	7	8	9	10	11
CP+AP (Regret)	3.0	11.2	8.9	119.0	314.0
CP+AP (SCN)	3.1	**7.6**	4	5.7	**9.7**
Enumeration	**0.4**	12.4	**0.2**	**0.8**	24.9
Ruland	12.8	62.1	62.8	85.7	399.0
Ruland+CP+AP	5.7	9.6	6.4	8.4	12.3
Sarin	54.1	273.0	141.0	760.0	1538.0
Sarin+CP+AP	16.3	23.2	24.9	41.3	50.4
	12	13	14	15	
CP+AP (Regret)	384.0	253.0	11,267.0	33,697	
CP+AP (SCN)	**8.5**	**84.2**	**45.2**	**155**	
Enumeration	-	-	-	-	
Ruland	189.0	690.0	1100.0	12,674	
Ruland+CP+AP	14.7	88.1	49.1	176	
Sarin	2271.0	3450.0	7671.0	49,416	
Sarin+CP+AP	64.3	355.0	133.0	12,972	

Table 3 Median milliseconds to find a solution within 10% of optimal.

Solver	Pickup and Delivery Pairs 2	3	4	5	6
CP+AP (Regret)	0.9	1.4	2.6	**4.8**	**14.9**
CP+AP (SCN)	0.9	1.3	3.5	5.4	28.9
Enumeration	**0.2**	**0.4**	**2.4**	24.7	595.0
Ruland	2.9	4.2	6.3	8.6	18.8
Ruland+CP+AP	2.5	3.0	4.6	6.6	30.2
Sarin	2.7	6.0	12.6	42.2	51.9
Sarin+CP+AP	2.5	4.1	6.4	11.2	34.2
	7	8	9	10	11
CP+AP (Regret)	**23.5**	109.0	230.0	1236	-
CP+AP (SCN)	43.6	353.0	1015.0	2268	-
Enumeration	11,868.0	-	-	-	-
Ruland	31.8	**71.1**	**70.4**	**129**	10,263
Ruland+CP+AP	47.0	178.0	176.0	239	**4839**
Sarin	236.0	669.0	724.0	2283	39,822
Sarin+CP+AP	59.2	641.0	804.0	3011	38,810
	12	13	14	15	
CP+AP (Regret)	14,533	-	-	-	
CP+AP (SCN)	90,266	-	-	-	
Enumeration	-	-	-	-	
Ruland	511	6868	-	-	
Ruland+CP+AP	**481**	**5007**	**6598**	-	
Sarin	31,017	77,622	-	-	
Sarin+CP+AP	41,130	61,273	-	-	

Table 5 Median milliseconds to prove optimality.

Hybrid CP is an effective optimizer for TSPPD instances with up to 8 pairs, after which point MIP is the dominant technique. The Ruland MIP model by itself loses the ability to find feasible solutions quickly as problem size increases. A second hybridization of warm starting MIP using the best solution found by the CP solver within 100 ms handles this problem nicely, and can help the Ruland model find optimal solutions and prove optimality faster. The Sarin model finds feasible solutions quickly, but has the disadvantage of additional overhead due to model formulation.

For the range of problems that we are looking at, we find Held-Karp bounds ineffective at reducing solution time due to a large optimality gap in the face of precedence constraints. On the other hand, we observe dramatic improvements to the abilities of CP models to optimize from incorporating incremental primal-dual AP relaxations and reduced cost-base variable domain filtering.

When TSPPD problems are small, most models are effective for fast solution. It appears that one loses little by choosing an algorithm that performs well on instance with more than 10 pairs and using it on the smaller ones as well. The Ruland model is the best at finding and proving optimal solutions, but requires the problem instances be symmetric. Difficulties attaining feasibility of the Ruland model are easily overcome through warm starting, and a sophisticated warm start mechanism such as Hybrid CP with SCN branching can positively impact its optimization abilities as well. For asymmetric TSPPD instances, the Sarin model can be effectively warm started in the same manner.

We believe these results provide further evidence for the utility of hybrid optimization methods in solving hard combinatorial problems with side constraints, and demonstrate their utility in real-time optimization. These studies can be extended to incorporate additional side constraints found in common pickup and delivery problems, such as capacity and time windows. We expect such exploration will further demonstrate the benefits of hybridization.

6 Compliance with Ethical Standards

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Aguayo, M.M., Sarin, S.C., Sherali, H.D.: Solving the single and multiple asymmetric traveling salesmen problems by generating subtour elimination constraints frominteger solutions. IISE Transactions **50**(1), 45–53 (2018)
2. Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The traveling salesman problem: a computational study. Princeton university press (2011)
3. Benchimol, P., Régin, J.C., Rousseau, L.M., Rueher, M., van Hoeve, W.J.: Improving the held and karp approach with constraint programming. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, pp. 40–44. Springer (2010)
4. Benchimol, P., Van Hoeve, W.J., Régin, J.C., Rousseau, L.M., Rueher, M.: Improved filtering for weighted circuit constraints. Constraints **17**(3), 205–233 (2012)
5. Berbeglia, G., Cordeau, J.F., Laporte, G.: A hybrid tabu search and constraint programming algorithm for the dynamic dial-a-ride problem. INFORMS Journal on Computing **24**(3), 343–355 (2012)
6. Berbeglia, G., Hahn, G.: Counting feasible solutions of the traveling salesman problem with pickups and deliveries is #P-complete. Discrete Applied Mathematics **157**(11), 2541–2547 (2009)
7. Bertsimas, D., Jaillet, P., Martin, S.: Online Vehicle Routing: The Edge of Optimization in Large-Scale Applications (2018). Accepted for publication in Operations Research
8. Carpaneto, G., Martello, S., Toth, P.: Algorithms and codes for the assignment problem. Annals of operations research **13**(1), 191–223 (1988)
9. Caseau, Y., Laburthe, F.: Solving Small TSPs with Constraints. In: ICLP, vol. 97, p. 104 (1997)
10. Dantzig, G., Fulkerson, R., Johnson, S.: Solution of a large-scale traveling-salesman problem. Journal of the operations research society of America **2**(4), 393–410 (1954)
11. Dumas, Y., Desrosiers, J., Soumis, F.: The pickup and delivery problem with time windows. European journal of operational research **54**(1), 7–22 (1991)
12. Dumitrescu, I., Ropke, S., Cordeau, J.F., Laporte, G.: The traveling salesman problem with pickup and delivery: polyhedral results and a branch-and-cut algorithm. Mathematical Programming **121**(2), 269 (2010)
13. Edmonds, J.: Optimum branchings. Journal of Research of the National Bureau of Standards B **71**(4), 233–240 (1967)
14. Fischetti, M., Toth, P.: An additive bounding procedure for the asymmetric travelling salesman problem. Mathematical Programming **53**(1-3), 173–197 (1992)
15. Fisher, M.L.: The lagrangian relaxation method for solving integer programming problems. Management science **50**(12_supplement), 1861–1871 (2004)
16. Focacci, F., Lodi, A., Milano, M.: A hybrid exact algorithm for the TSPTW. INFORMS Journal on Computing **14**(4), 403–417 (2002)
17. Focacci, F., Lodi, A., Milano, M., Vigo, D.: Solving TSP through the integration of OR and CP techniques. Electronic notes in discrete mathematics **1**, 13–25 (1999)
18. Gabow, H.N., Galil, Z., Spencer, T., Tarjan, R.E.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica **6**(2), 109–122 (1986)
19. Gecode Team: Gecode: Generic constraint development environment (2006). URL `http://www.gecode.org`
20. Grubhub: TSPPD Test Instance Library. `https://github.com/grubhub/tsppdlib` (2018)

21. Gurobi Optimization, Inc.: Gurobi optimizer reference manual (2016). URL http://www.gurobi.com
22. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees. Operations Research **18**(6), 1138–1162 (1970)
23. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees: Part ii. Mathematical programming **1**(1), 6–25 (1971)
24. Lauriere, J.L.: A language and a program for stating and solving combinatorial problems. Artificial intelligence **10**(1), 29–127 (1978)
25. Law, Y., Lee, J.: Global constraints for integer and set value precedence. Principles and Practice of Constraint Programming–CP 2004 pp. 362–376 (2004)
26. Lin, S.: Computer solutions of the traveling salesman problem. The Bell System Technical Journal **44**(10), 2245–2269 (1965)
27. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Operations research **21**(2), 498–516 (1973)
28. van Lon, R.R., Ferrante, E., Turgut, A.E., Wenseleers, T., Berghe, G.V., Holvoet, T.: Measures of dynamism and urgency in logistics. European Journal of Operational Research **253**(3), 614–624 (2016)
29. Maher, S.J., Fischer, T., Gally, T., Gamrath, G., Gleixner, A., Gottwald, R.L., Hendel, G., Koch, T., Lübbecke, M.E., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Weninger, D., Witt, J.T., Witzig, J.: The scip optimization suite 4.0. Tech. Rep. 17-12, ZIB, Takustr.7, 14195 Berlin (2017)
30. Milano, M., Ottosson, G., Refalo, P., Thorsteinsson, E.S.: Global constraints: When constraint programming meets operation research. In: INFORMS Journal on Computing, Special Issue on the Merging of Mathematical Programming and Constraint Programming. Citeseer (2001)
31. Miller, C.E., Tucker, A.W., Zemlin, R.A.: Integer programming formulation of traveling salesman problems. Journal of the ACM **7**(4), 326–329 (1960)
32. O'Neil, R.J.: TSPPD Hybrid Optimization Code. https://github.com/ryanjoneil/tsppd-hybrid (2018)
33. Padberg, M., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM review **33**(1), 60–100 (1991)
34. Papadimitriou, C.H.: The Euclidean travelling salesman problem is NP-complete. Theoretical computer science **4**(3), 237–244 (1977)
35. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transportation Science **32**(1), 12–29 (1998)
36. Pferschy, U., Staněk, R.: Generating subtour elimination constraints for the TSP from pure integer solutions. Central European Journal of Operations Research pp. 1–30 (2016)
37. Psaraftis, H.N.: A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. Transportation Science **14**(2), 130–154 (1980)
38. Ruland, K., Rodin, E.: The pickup and delivery problem: Faces and branch-and-cut algorithm. Computers & mathematics with applications **33**(12), 1–13 (1997)
39. Sarin, S.C., Sherali, H.D., Bhootra, A.: New tighter polynomial length formulations for the asymmetric traveling salesman problem with and without precedence constraints. Operations research letters **33**(1), 62–70 (2005)
40. Valenzuela, C.L., Jones, A.J.: Estimating the held-karp lower bound for the geometric tsp. European journal of operational research **102**(1), 157–175 (1997)
41. Wickham, H.: ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York (2009). URL http://ggplot2.org

7 Appendix

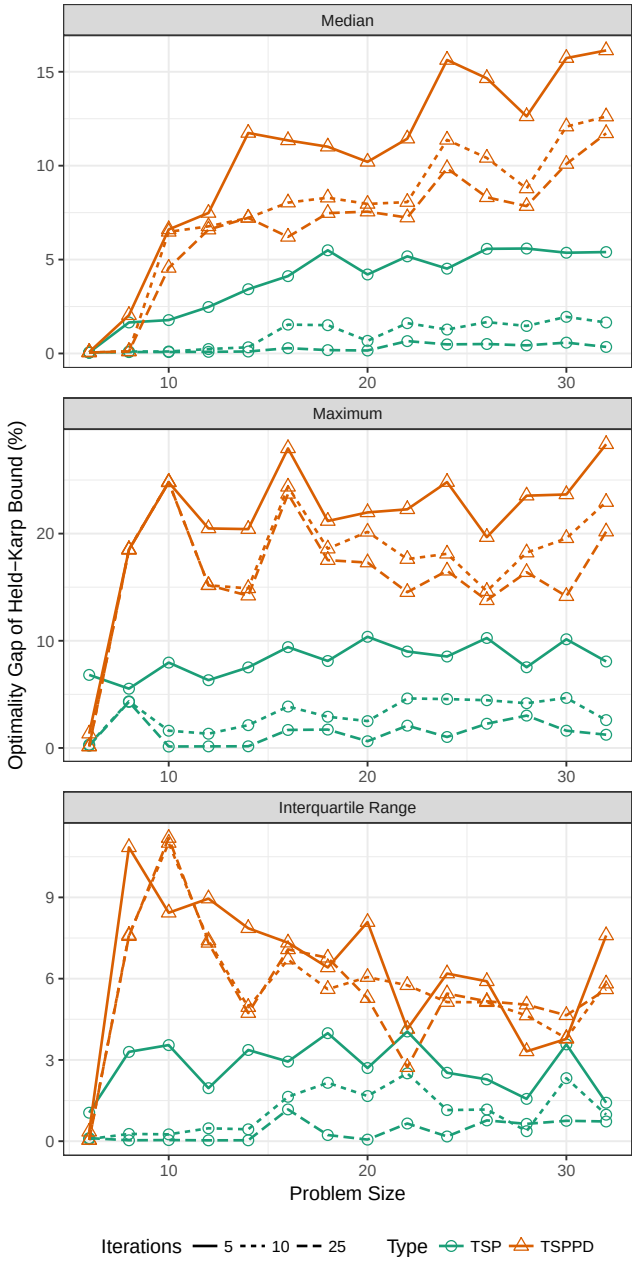

Fig. 15 Optimality gap of Held-Karp bounds.

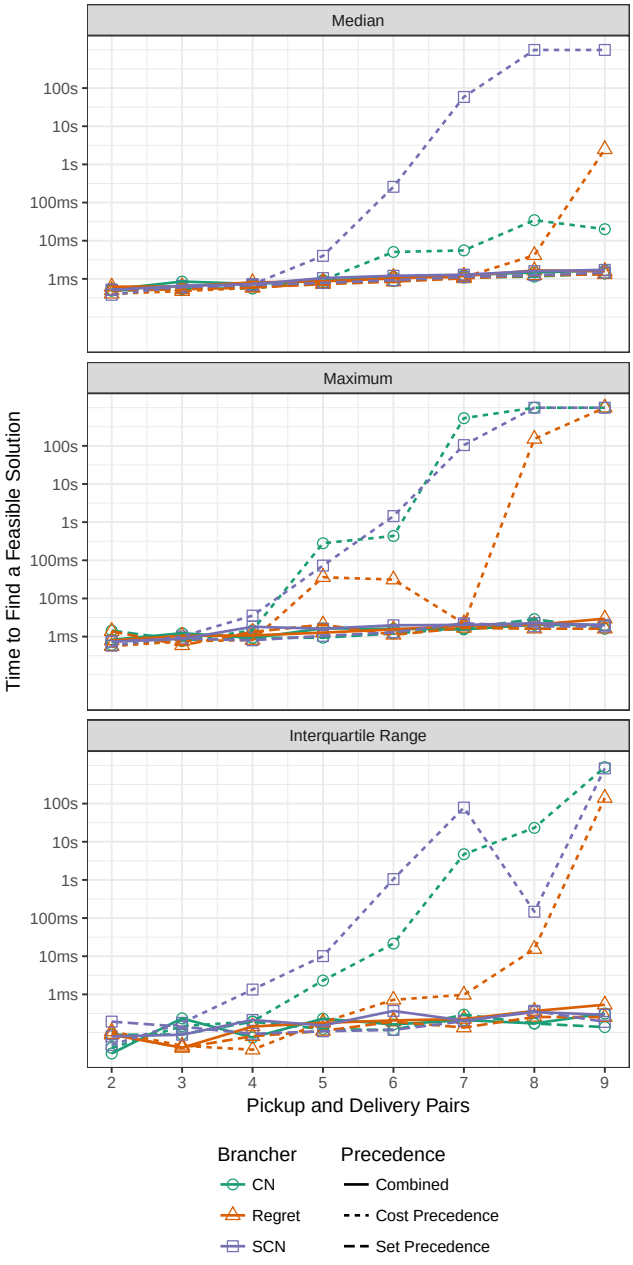

Fig. 16 Time to find a feasible solution for precedence forms in TSPPD CP models.

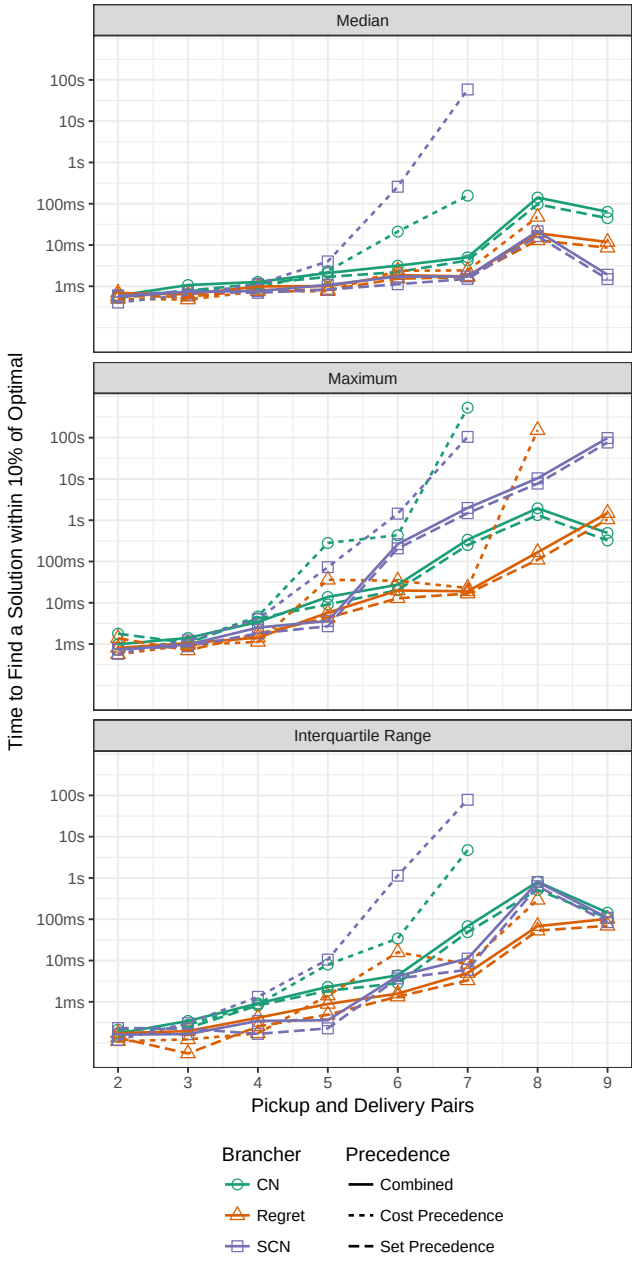

Fig. 17 Time to find a solution within 10% of optimal for precedence forms in TSPPD CP models.

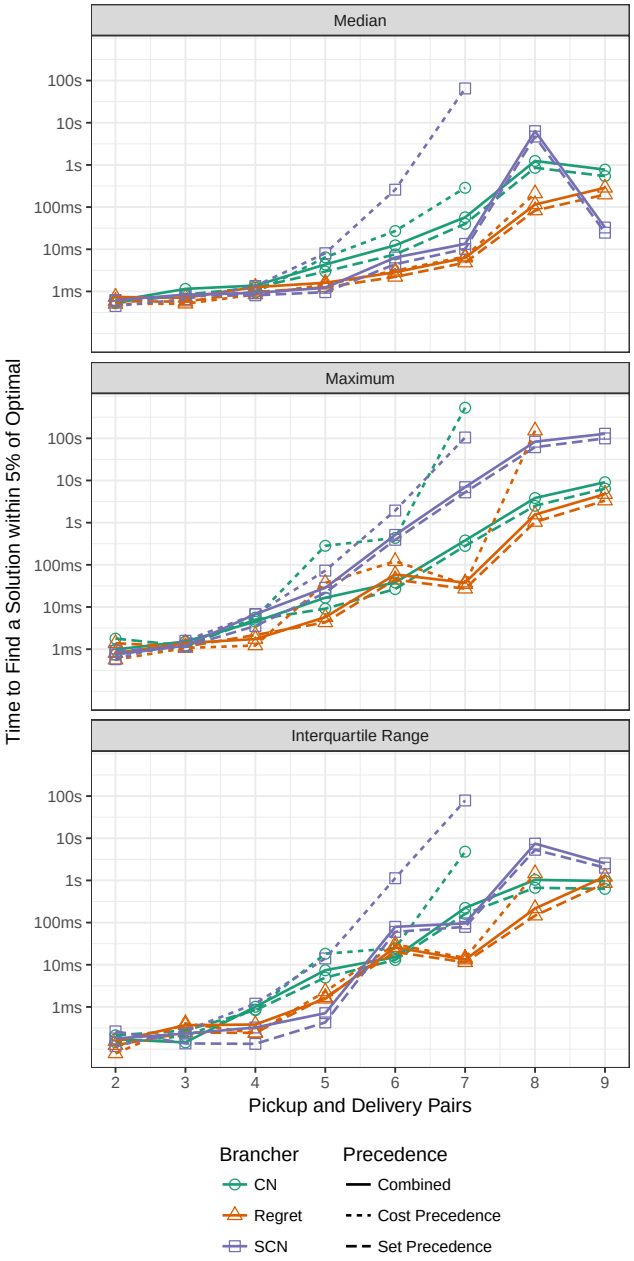

Fig. 18 Time to find a solution within 5% of optimal for precedence forms in TSPPD CP models.

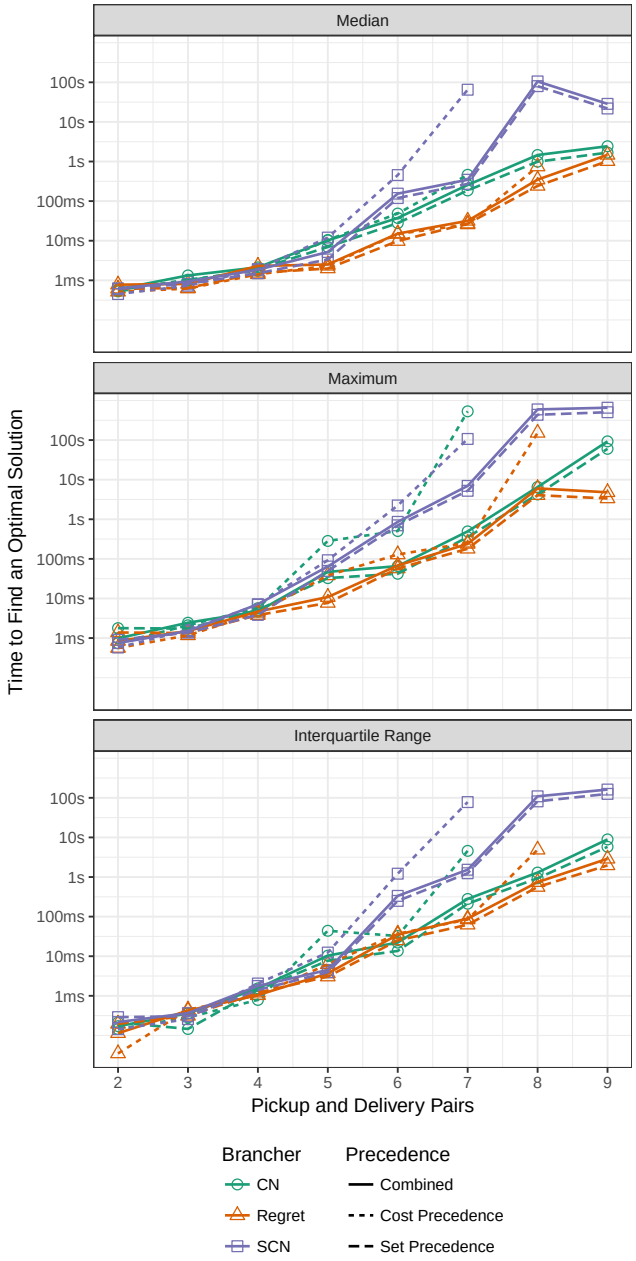

Fig. 19 Time to find an optimal solution for precedence forms in TSPPD CP models.

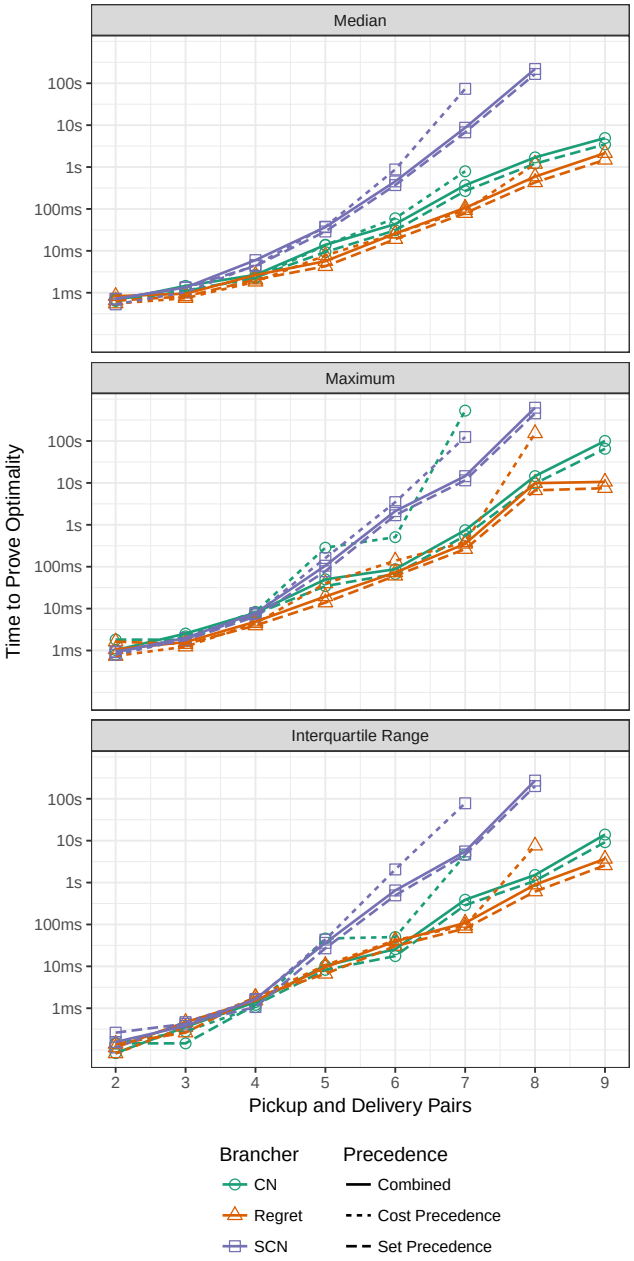

Fig. 20 Time to prove optimality for precedence forms in TSPPD CP models.

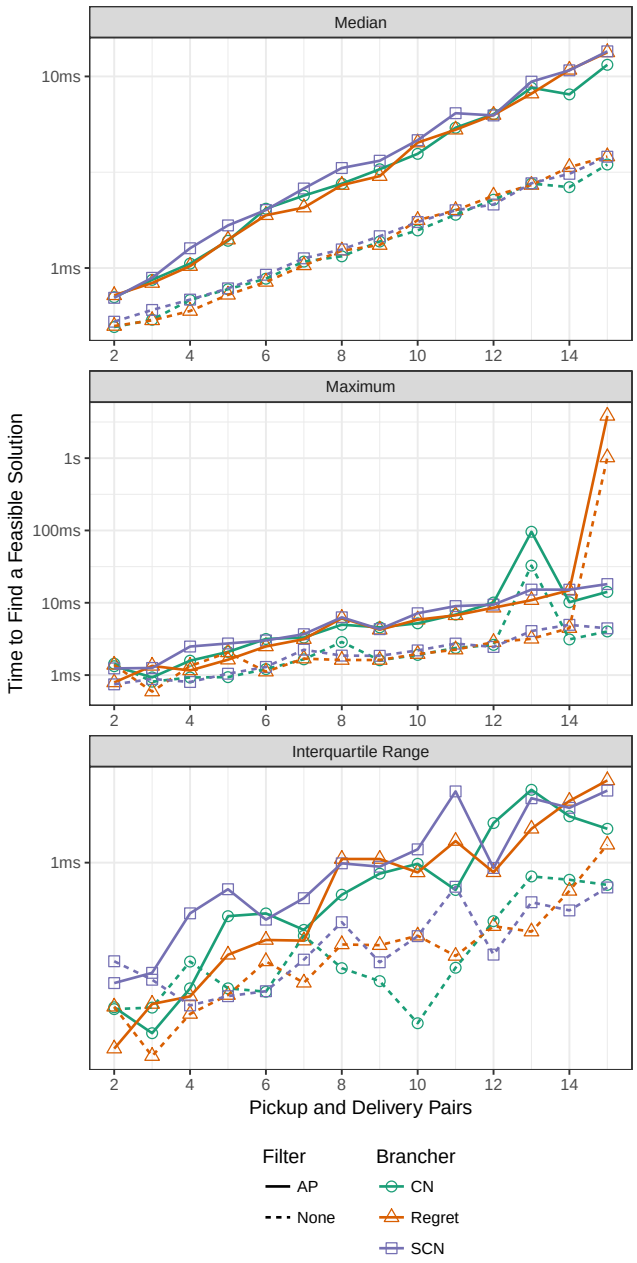

Fig. 21 Time to find a feasible solution for TSPPD CP models with and without AP filtering.

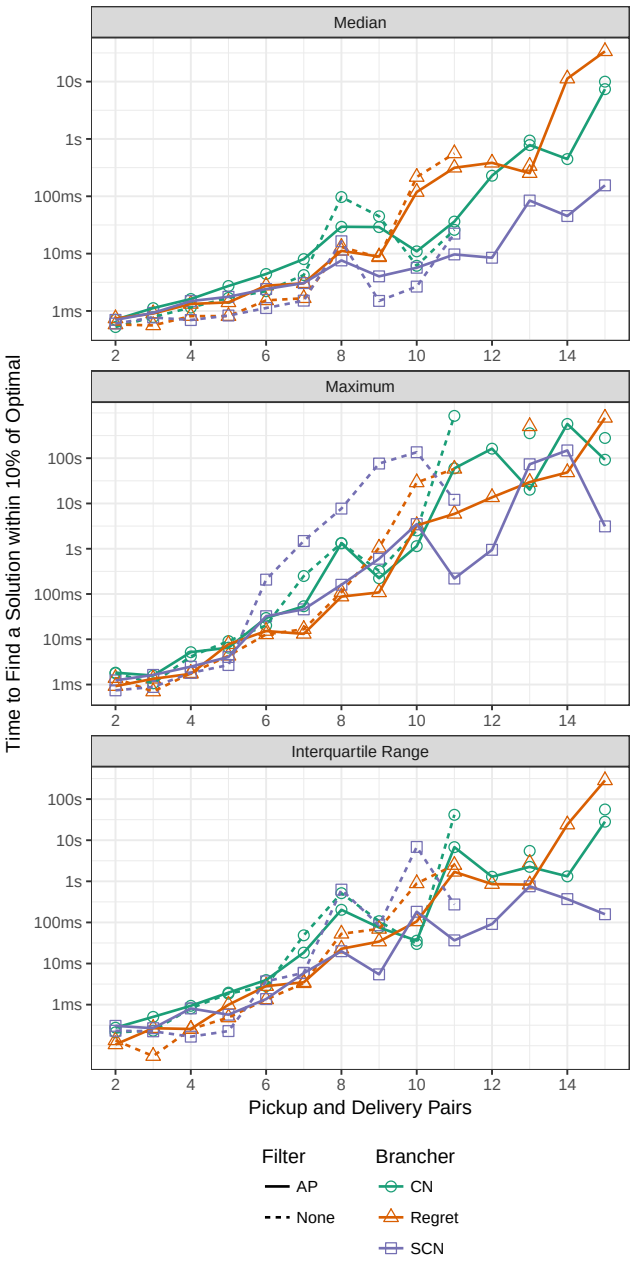

Fig. 22 Time to find a solution within 10% of optimal for TSPPD CP models with and without AP filtering.

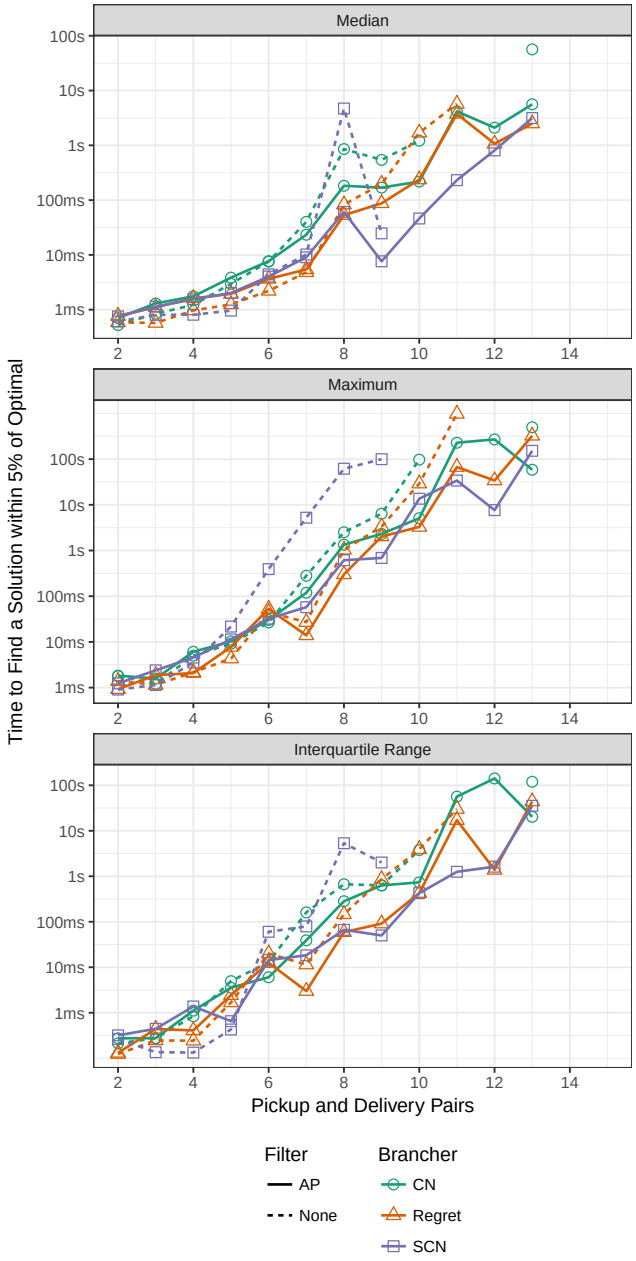

Fig. 23 Time to find a solution within 5% of optimal for TSPPD CP models with and without AP filtering.

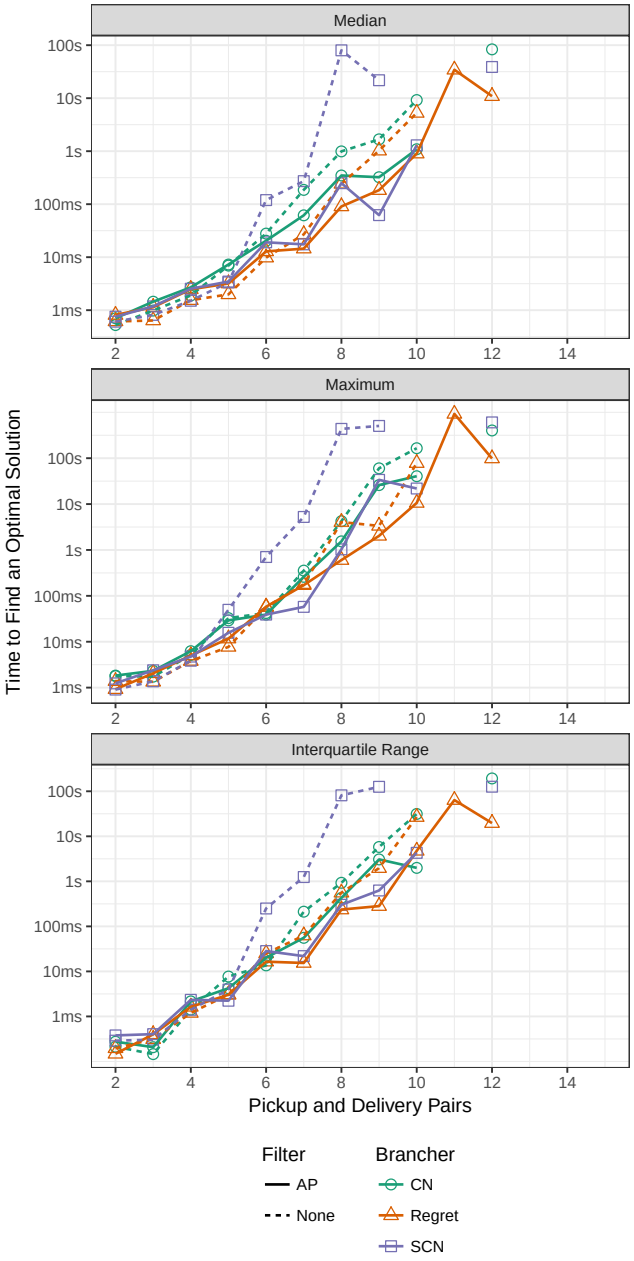

Fig. 24 Time to find an optimal solution for TSPPD CP models with and without AP filtering.

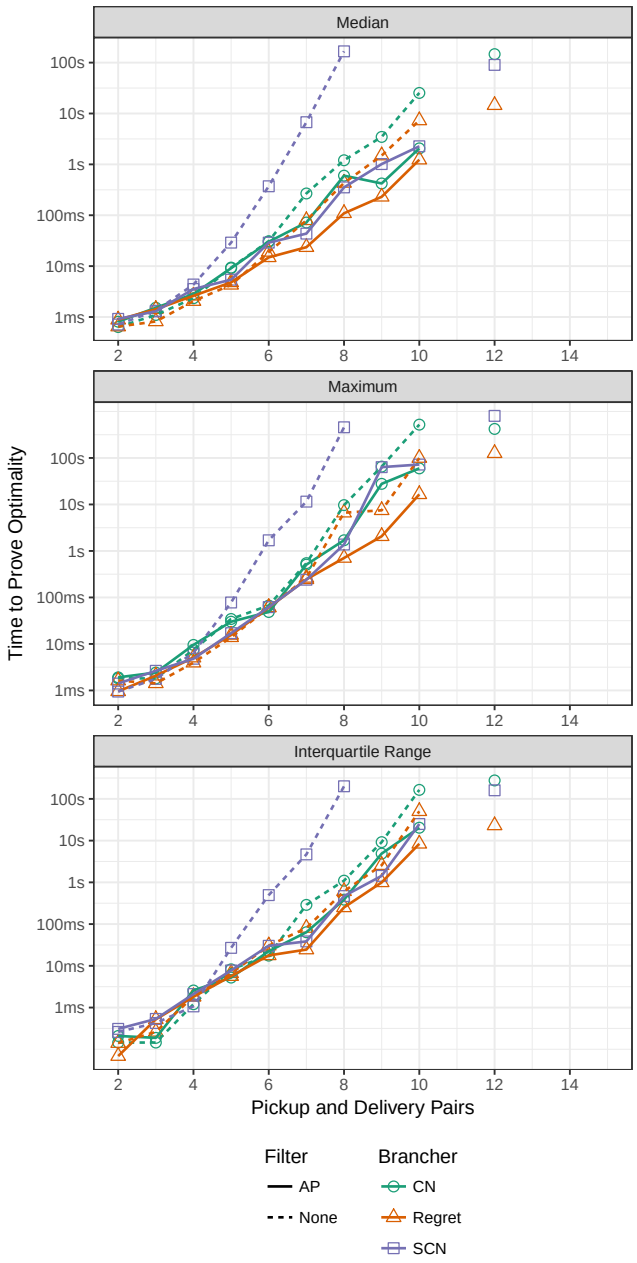

Fig. 25 Time to prove optimality for TSPPD CP models with and without AP filtering.

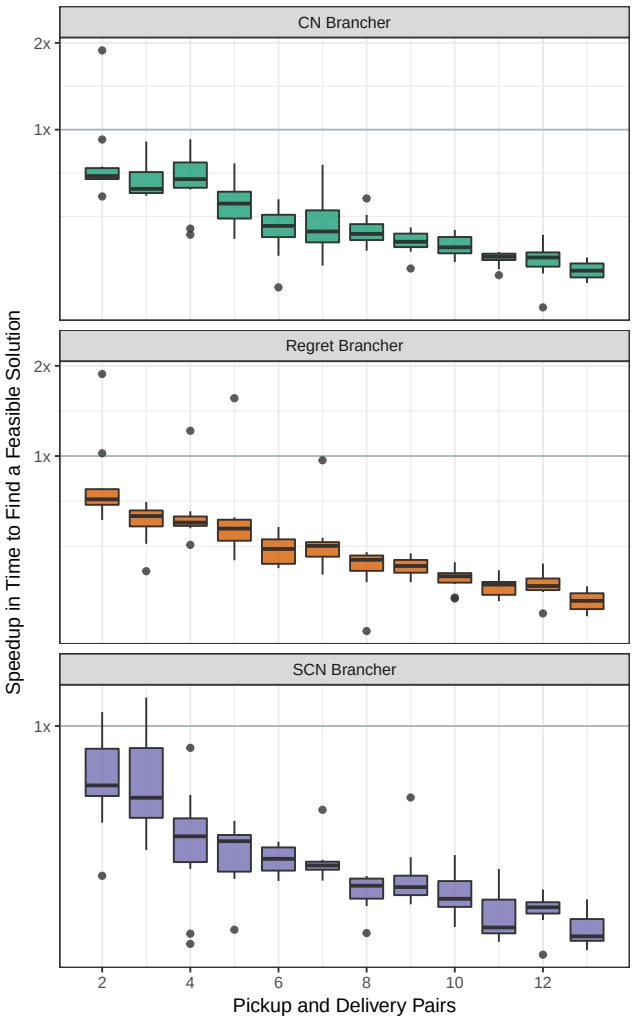

Fig. 26 Speedup finding feasible solutions due to AP relaxation and reduced cost fixing.

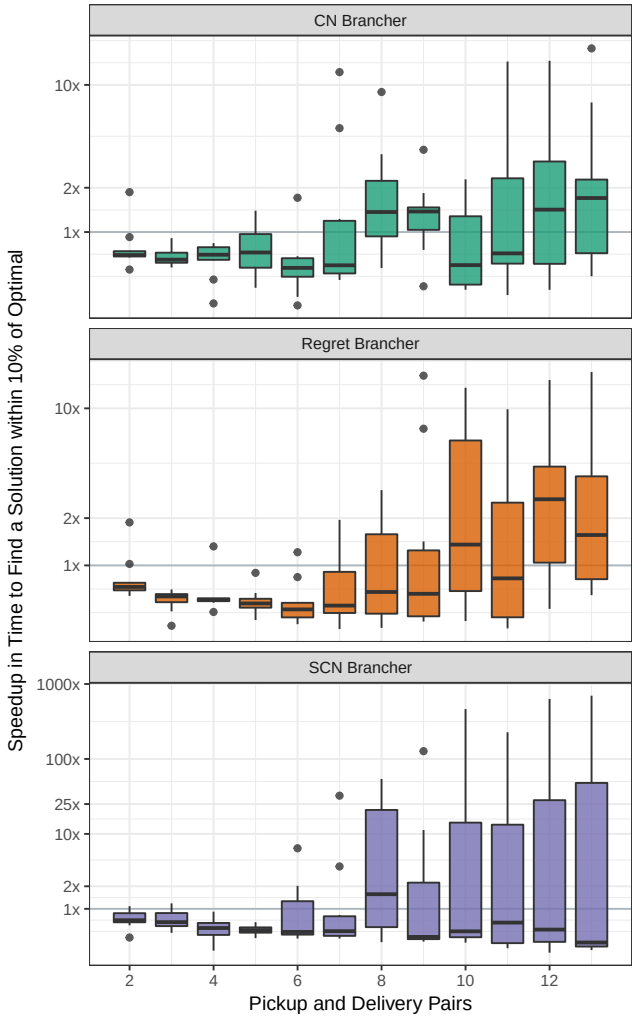

Fig. 27 Speedup finding solutions within 10% of optimal due to AP relaxation and reduced cost fixing.

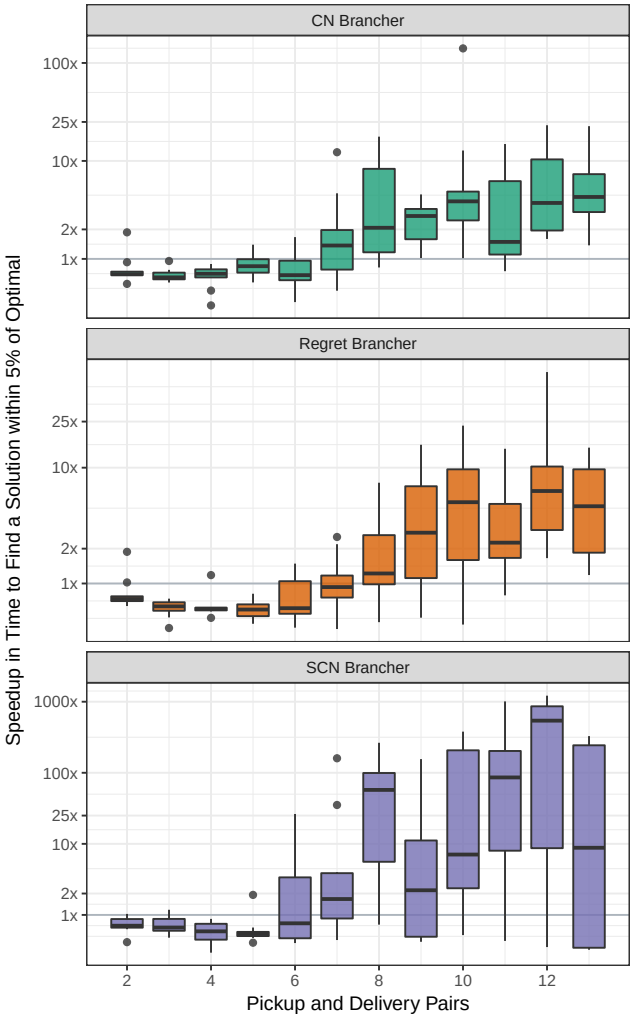

Fig. 28 Speedup finding solutions within 5% of optimal due to AP relaxation and reduced cost fixing.

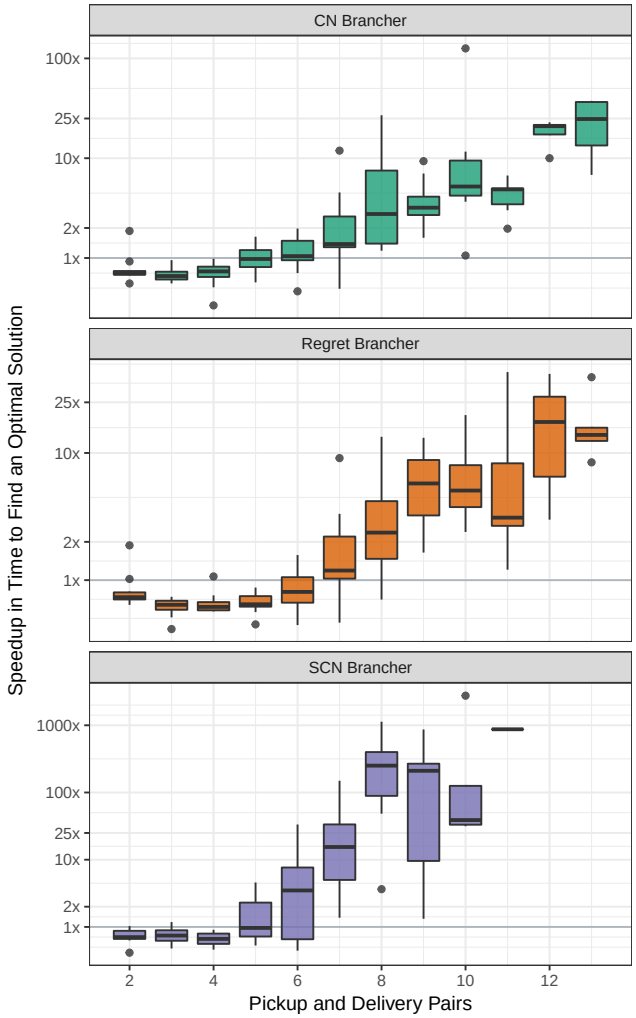

Fig. 29 Speedup finding optimal solutions due to AP relaxation and reduced cost fixing.

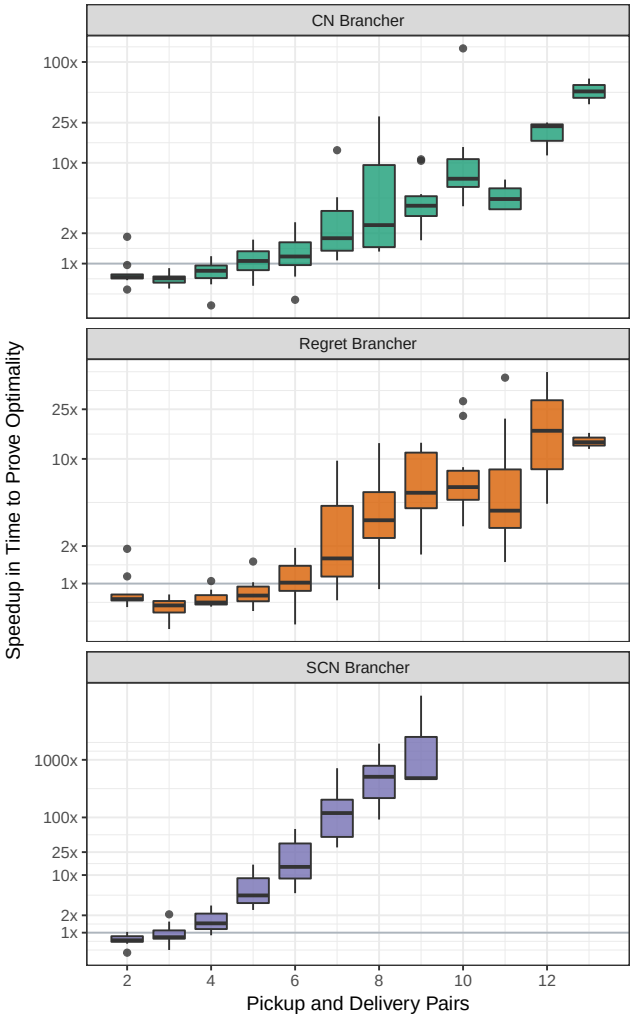

Fig. 30 Speedup proving optimality due to AP relaxation and reduced cost fixing.

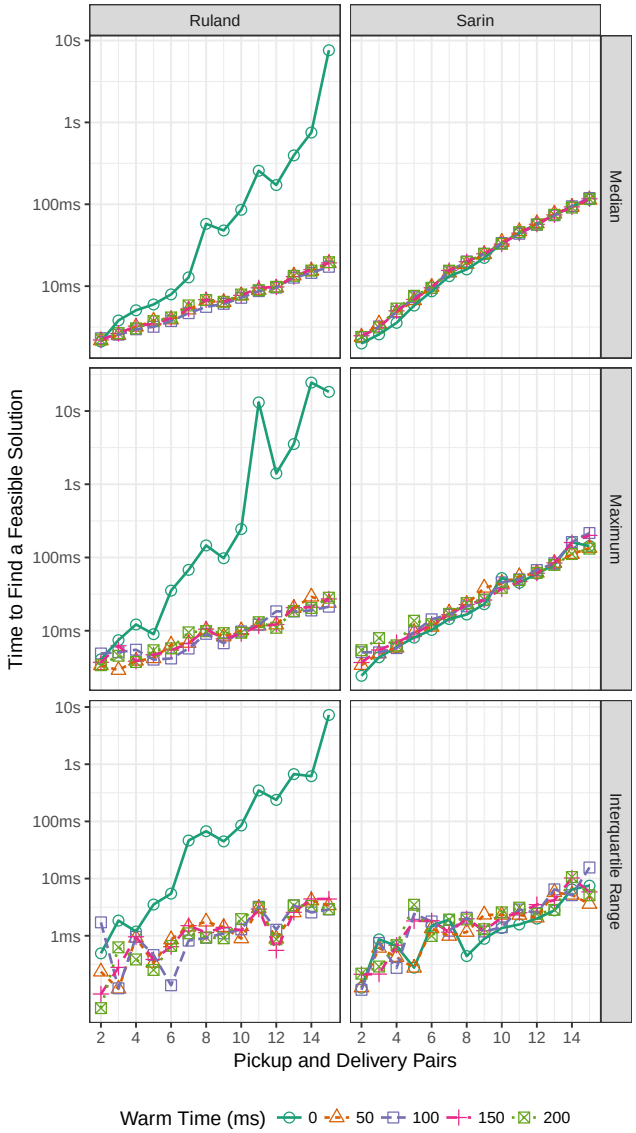

Fig. 31 Time to find a feasible solution with a Hybrid CP warm start.

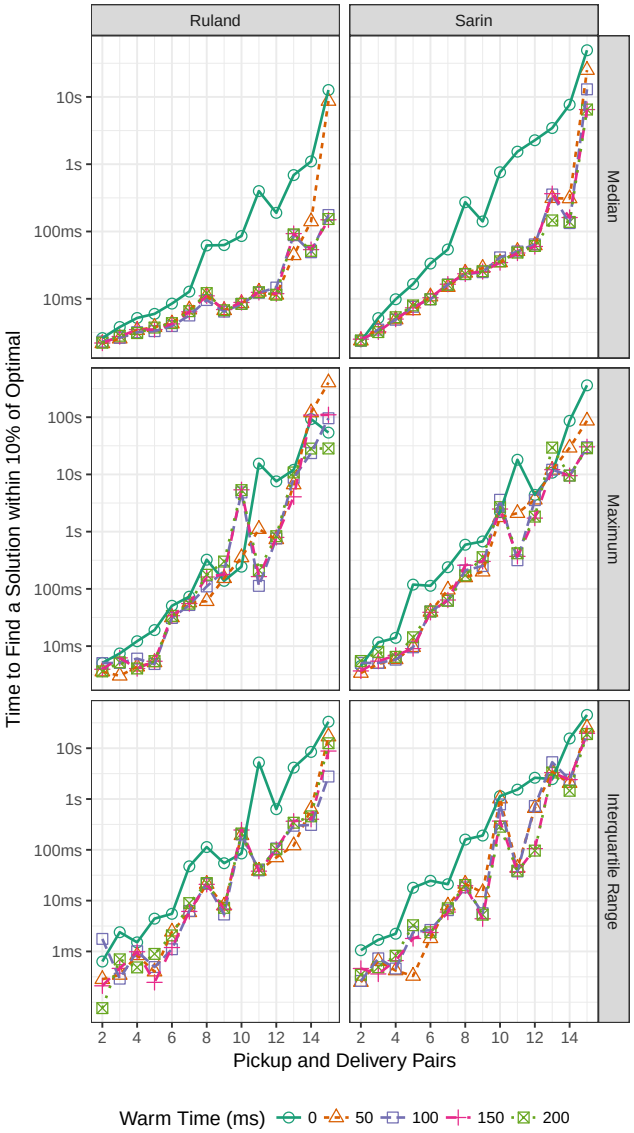

Fig. 32 Time to find a solution within 10% of optimal with a Hybrid CP warm start.

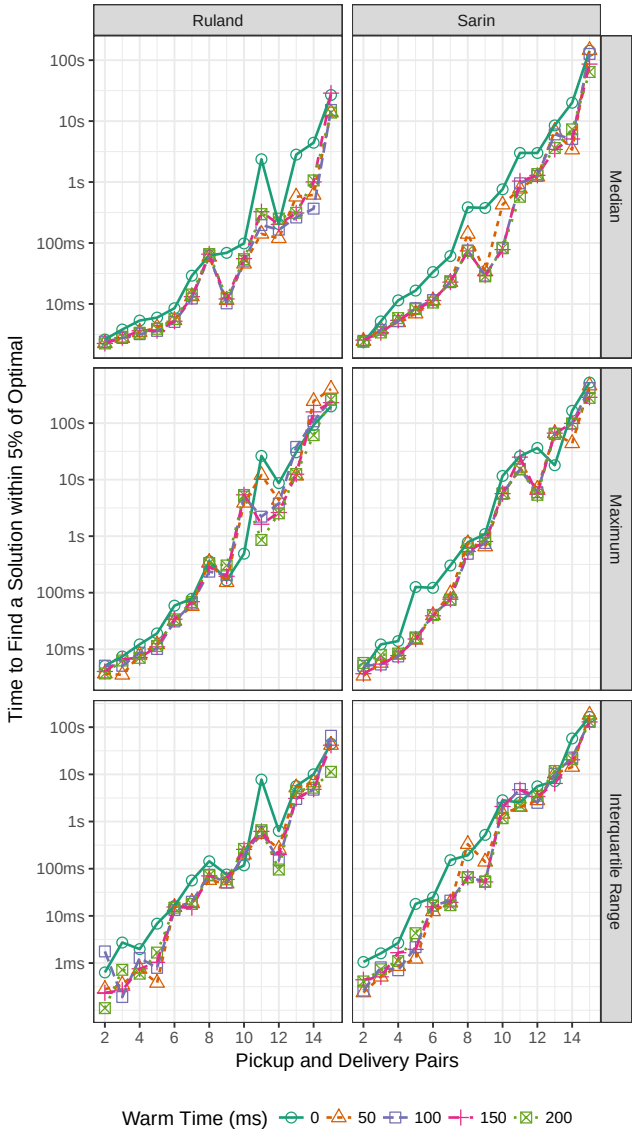

Fig. 33 Time to find a solution within 5% of optimal with a Hybrid CP warm start.

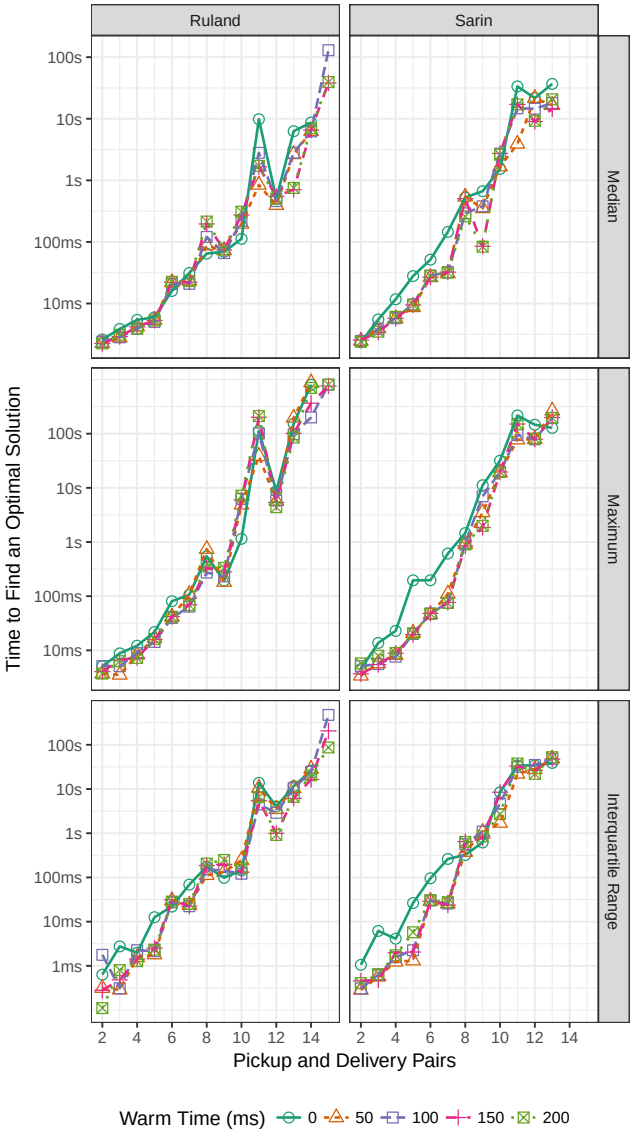

Fig. 34 Time to find an optimal solution with a Hybrid CP warm start.

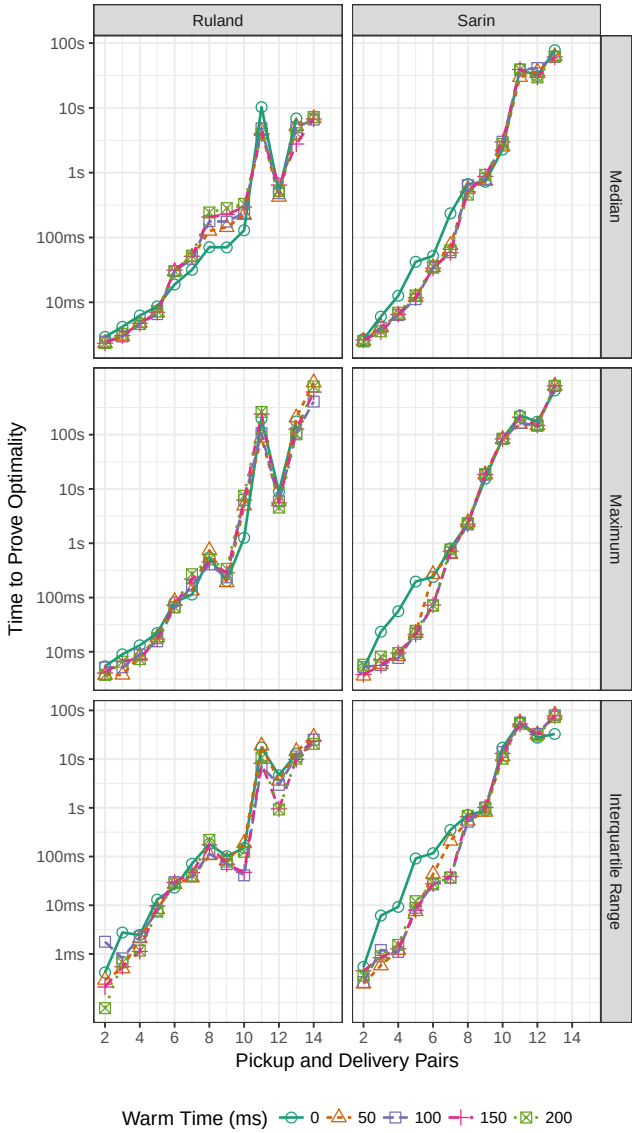

Fig. 35 Time to prove optimality with a Hybrid CP warm start.

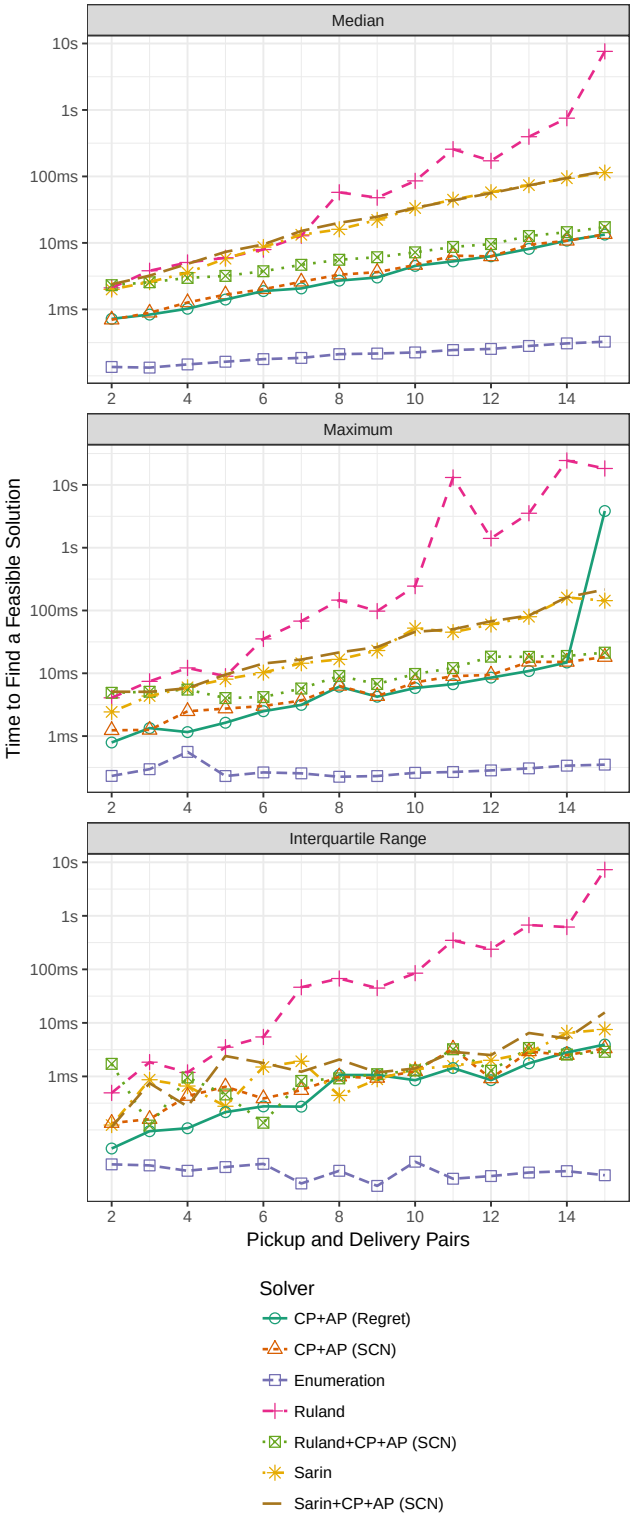

Fig. 36 Time to find a feasible solution for different solution techniques.

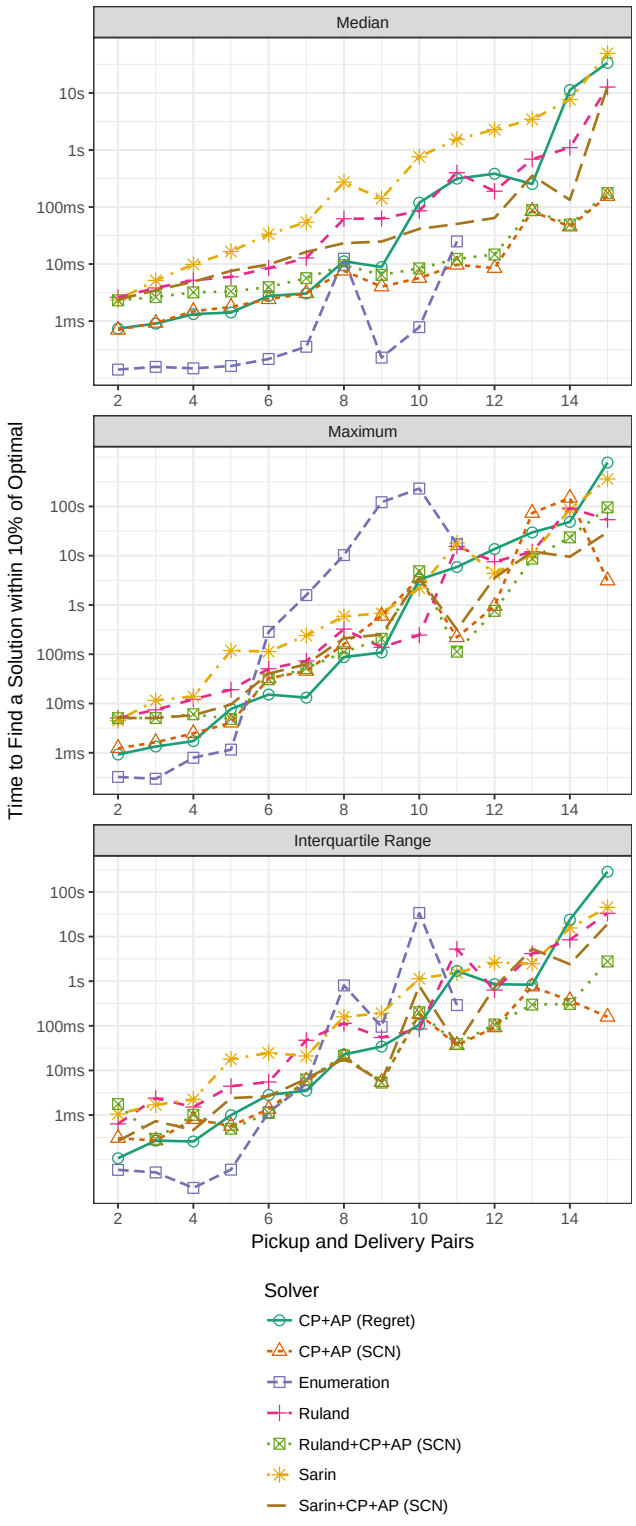

Fig. 37 Time to find a solution within 10% of optimal for different solution techniques.

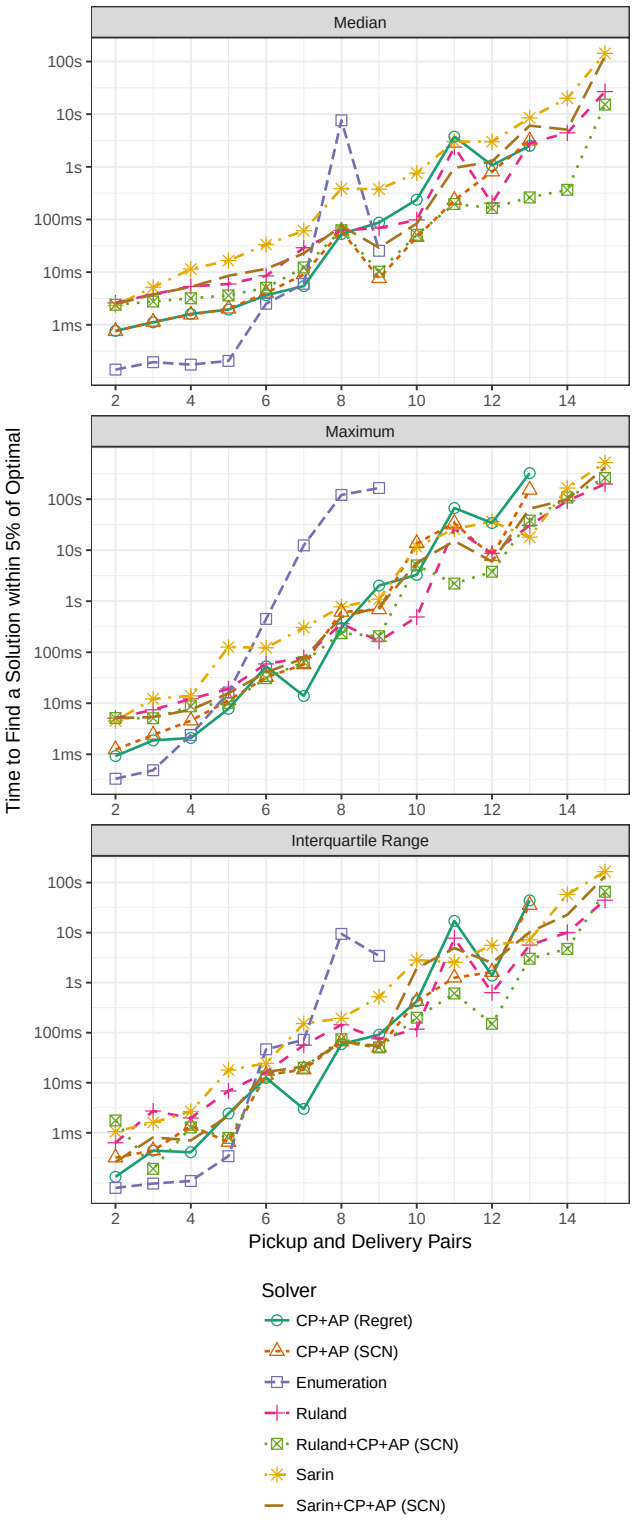

Fig. 38 Time to find a solution within 5% of optimal for different solution techniques.

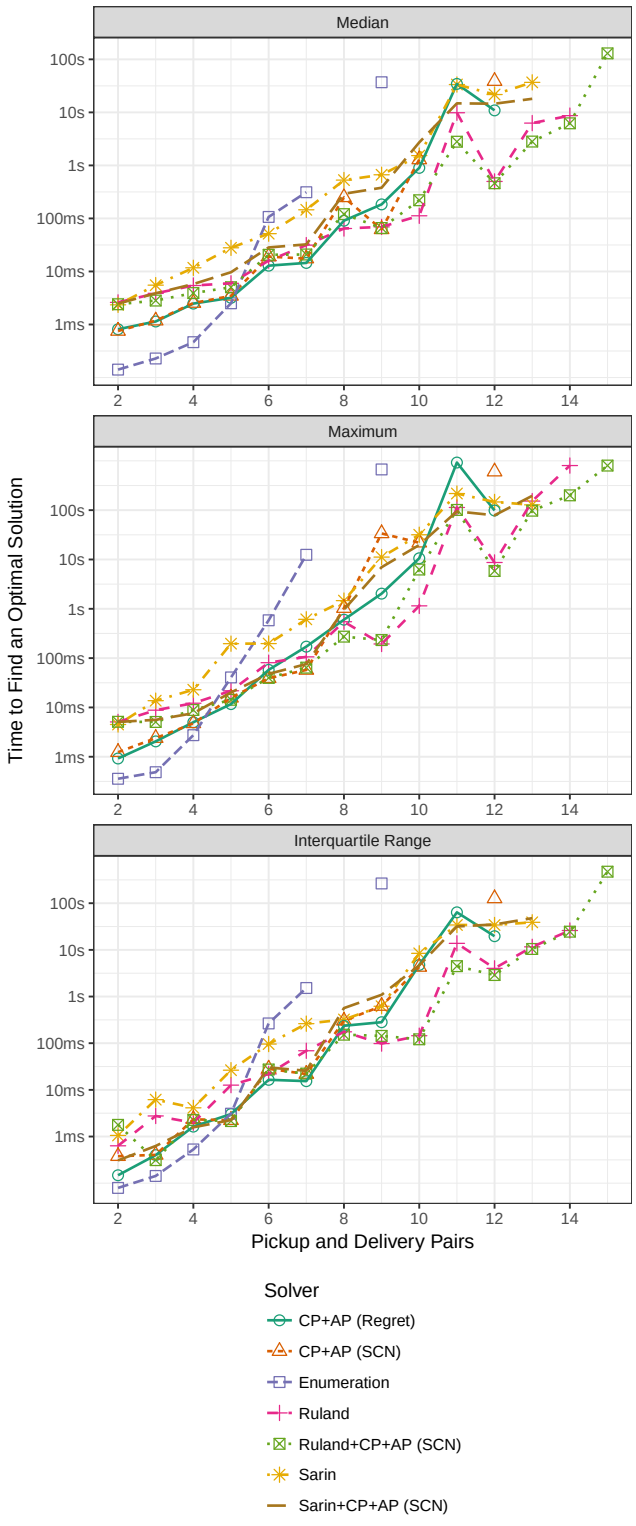

Fig. 39 Time to find an optimal solution for different solution techniques.

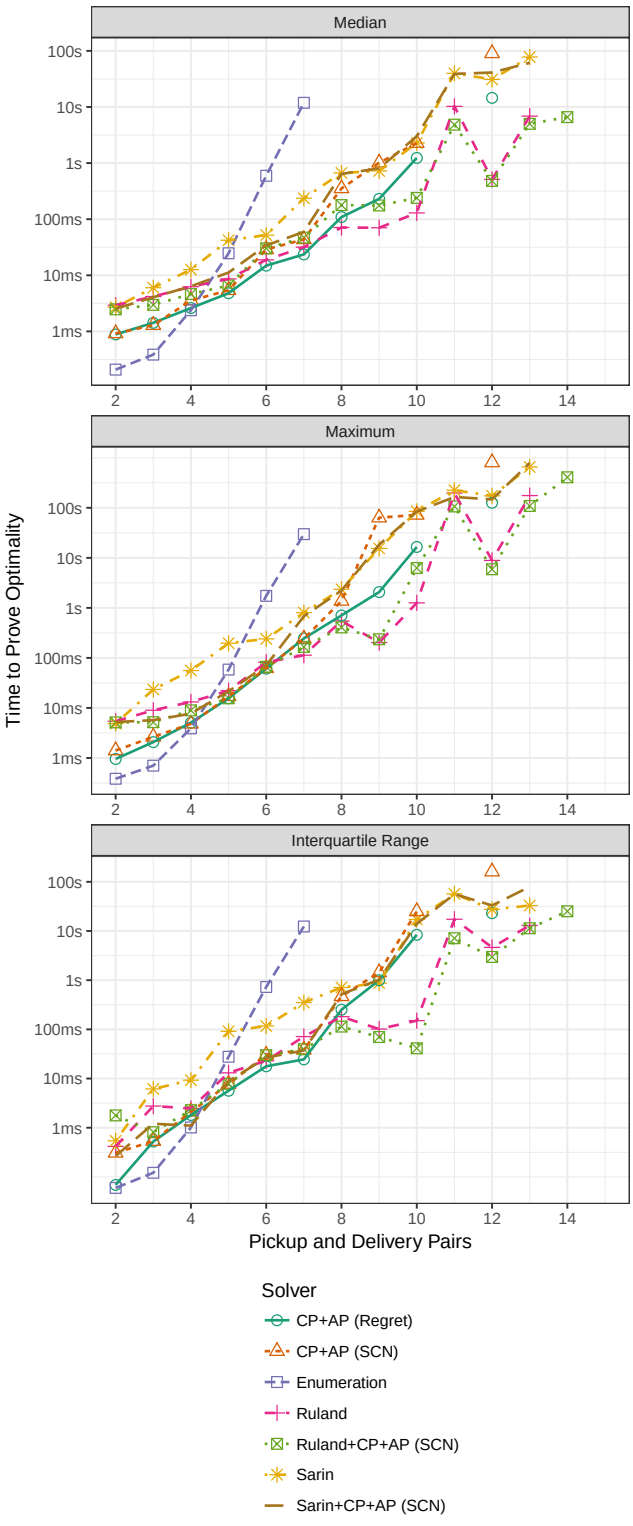

Fig. 40 Time to prove optimality for different solution techniques.