Under Review.

Column Elimination: An Iterative Approach to Solving Arc-Flow Formulations

Anthony Karahalios
Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, akarahal@andrew.cmu.edu
Willem-Jan van Hoeve
Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, vanhoeve@andrew.cmu.edu

We present column elimination as a general framework for solving problems whose solutions can be represented as a collection of decision sequences. We show how such problems can be modeled as arc-flow formulations, using a dynamic program to represent the set of feasible decision sequences and their costs, and an integer linear program over the corresponding state-transition graph. Because the exact arc-flow formulation can be exponentially large, we introduce relaxed dynamic programs to compactly represent a superset of the feasible decision sequences. Column elimination solves the exact model by starting with a relaxed dynamic program, which induces a relaxed arc-flow formulation, and iteratively refining the relaxation by eliminating infeasible sequences in optimal solutions. Doing so, we generalize earlier approaches that were developed for decision-diagram based network models. We apply our general column elimination framework to three new applications: the vehicle routing problem with time windows, the pickup-and-delivery problem with time windows, and the graph multicoloring problem. The computational results show that column elimination is competitive with or outperforms state-of-the-art methods, closing several previously open benchmark instances across these domains, including one vehicle routing problem with time windows on 1,000 locations.

Key words: integer programming, column elimination, column generation, dynamic programming, network flows, arc-flow formulations

History: Submitted on XXX

1. Introduction

The computational revolution in integer programming solvers over the last decades has enabled the ability to solve problems with hundreds of thousands of integer variables in reasonable time.

It has expanded the application of this powerful technology from strategic planning problems to detailed operational decision making and even real-time use cases. For several important problem domains, however, general integer programming does not scale to the requirements demanded by the application. Examples include vehicle routing problems such as last-mile delivery, complex multi-machine scheduling applications, and airline crew scheduling. In such cases alternative methods including Benders decomposition, branch-and-price, or constraint programming can be more effective, providing a different problem representation and associated solution methodology.

In this work, we present *column elimination* as a general solution framework that integrates ideas from dynamic programming, decision diagrams, network flows, and linear and integer programming. The starting point of the framework is a problem representation that is similar to that of *column generation*, i.e., in which a variable (or a column) represents a specific combinatorial structure such as a route or a schedule. A column formulation lists all possible variables and then selects an optimal subset of columns that satisfies the constraints. Because column formulations are, in general, exponentially large, column elimination starts by representing a *relaxation* of the columns. While this may seem counter-intuitive, the relaxation can be represented compactly as a directed acyclic graph which allows solving a polynomial-sized problem that implicitly represents an exponential number of columns. In this representation, a column corresponds to a path in the graph. Because we work with a relaxation, the solution may contain infeasible columns, or paths, which are iteratively removed from the graph until an optimal feasible solution is found.

This iterative method was first introduced by van Hoeve (2020, 2022) as an alternative to branch-and-price to solve the graph coloring problem; Karahalios and van Hoeve (2022) presented an improved variant that uses a portfolio of variable orderings to construct the directed acyclic graph. The method was subsequently applied to compute dual bounds for the traveling salesperson problem with a drone (Tang 2021, Tang and van Hoeve 2024). That work also introduced a subgradient descent method to solve the linear programs more efficiently. The term ‘column elimination’ was first mentioned in (van Hoeve and Tang 2022) to describe the method and draw the parallel with column generation. Lastly, Karahalios and van Hoeve (2023) applied column elimination to find dual bounds for the capacitated vehicle routing problem, including the addition of cutting planes, reduced cost-based variable fixing, and an improved subgradient descent method.

Contributions We present a generalized framework of column elimination for solving (acyclic) arc-flow formulations, incorporating and formalizing the existing approaches. The formalization includes the introduction of *relaxed dynamic programs* that are similar to state-space relaxations but more general, and a generic *refinement algorithm* to strengthen a relaxed dynamic program by removing an infeasible sequence. We also propose a new algorithm that embeds column elimination in branch-and-bound. Using the generic framework, we apply column elimination to three new applications– the vehicle routing problem with time windows, the pickup-and-delivery problem with time windows, and the graph multicoloring problem. Lastly, we provide a computational evaluation of our framework and find that column elimination is competitive with or outperforms the state-of-the-art on these three new problem domains as it closes closes five open instances of the graph multicoloring problem, one open instance with 1,000 locations of the vehicle routing problem with time windows, and 14 open instances of the pickup-and-delivery problem with time windows, for the first time.

The paper is organized as follows. We start by discussing related work in Section 2. We then present the general discrete optimization problem setting to which we apply column elimination in Section 3. In Section 4, we describe the underlying model of column elimination, combining dynamic programming and integer programming. Section 5 introduces the iterative column elimination algorithm, including the extensions cut-and-refine and branch-and-refine. We present the subgradient method for solving large-scale problems in Section 6. Section 7 presents the three combinatorial problems that we use as a computational case study. The experimental results are presented in Section 8. We provide a summary and conclusion in Section 9.

2. Related Work

Column elimination shares many similarities with column generation, which is a well-established computational method for solving linear programming models. It was introduced by Ford and Fulkerson

(1958) to solve multi-commodity network flow problems and later generalized by Dantzig and Wolfe (1960) for solving linear programs. The extension to integer programming is done through branch-and-price, where column generation is embedded inside a branch-and-bound framework (Appelgren 1971, Barnhart et al. 1998, Lübbecke and Desrosiers 2005). Branch-and-price provides state-of-the-art results for many discrete optimization problems, including graph coloring (Mehrotra and Trick 1996, Held et al. 2012), scheduling (van Den Akker et al. 1999, Chen and Powell 1999, Leus and Kowalczyk 2016, Oliveira and Pessoa 2020, Bulhoes et al. 2020, Pessoa et al. 2010), and vehicle routing problems (Fukasawa et al. 2006, Baldacci et al. 2011b, Pecin et al. 2017, Pessoa et al. 2018, Mandal et al. 2023).

Column generation solves a restricted master problem that contains a subset of the possible variables. It uses the associated dual variables to solve a pricing problem to generate a new primal variable with a negative reduced cost that can improve the master problem. Potential implementation challenges of branch-and-price include stabilization strategies to handle dual degeneracy and the representation of (branching) cuts in the pricing problem (Vanderbeck 2005). As we will see later in more detail, column elimination does not solve a pricing problem, and avoids these issues as a result. On the other hand, column elimination solves network flow problems that may contain more (arc) variables than the analogous column generation model, which uses one variable per column. Column elimination also depends on the number of refinement iterations, as column generation depends on the number of pricing problems solved. The relative computational benefits are therefore problem dependent, but we show in Section 8 that column elimination provides state-of-the-art results on three problem domains that have also been tackled with column generation.

Many branch-and-price methods, especially in the context of vehicle routing, rely on dynamic programming for solving the pricing problem. As the associated state space can grow exponentially large, *state-space relaxations* of dynamic programs are often used in the pricing problem, which is equivalent to relaxing the set of columns in the linear program being solved by column generation, thus providing dual bounds. Our work is closely related to this approach, as we also define a relaxed set of the variables with a dynamic program. While column generation uses the dynamic program to generate new variables via the pricing problem, column elimination uses the dynamic program to directly define a model over the relaxed set of columns. We will discuss more similarities and differences in Section 4.

Arc-flow formulations have been used successfully to model problems over directed networks with solutions that are either a single path (de Lima et al. 2022, Boland et al. 2017, Lozano et al. 2022, Tang and van Hoeve 2024) or a collection of paths (Gouveia et al. 2019, van Hoeve 2022, Kowalczyk et al. 2024). A specific recent application is the use of decision diagrams to solve optimization problems, which involves arc-flow formulations, restrictions, and relaxations (Bergman et al. 2016, Ciré and van Hoeve 2013, Bergman and Ciré 2018); we refer to Castro et al. (2022) and van Hoeve (2024) for recent surveys.

Column elimination originated from the decision diagram literature. The original method was developed in the context of graph coloring, where relaxed binary decision diagrams were iteratively refined to produce dual bounds of increasing strength (van Hoeve 2020), and later to find integer optimal solutions (van Hoeve 2022, Karahalios and van Hoeve 2022). The method was next extended to multi-valued decision diagrams and applied to find dual bounds for routing applications (van Hoeve and Tang 2022, Tang and van Hoeve 2024, Karahalios and van Hoeve 2023). Those series of works introduced the connection with state-space relaxations for vehicle routing and developed more efficient algorithms for solving the constrained network flow problem over the decision diagram. This work formalizes and generalizes the previous works in two main ways. First, we formally define the structure of problems that column elimination can solve, and use dynamic programming models as an abstraction to define the network of solutions; we introduce the notion of relaxed dynamic programs to generalize relaxed decision diagrams and state-space relaxations that were used before. Second, we revisit the prior refinement algorithm, that relied on the respective network definitions, and generalize it to strengthen the relaxed dynamic program. In addition, this paper introduces a new algorithm that embeds column elimination in branch-and-bound. Lastly, we apply column elimination to three new applications: the vehicle routing problem with time windows, the pickup-and-delivery problem with time windows, and the graph multicoloring problem.

There are four related methods that solve a series of strengthened relaxations. First, dynamic discretization discovery is a framework for solving network design problems (Boland et al. 2017) This method solves an arc-flow formulation by strengthening discrete relaxations, but creates relaxations only by discretizing state values such as time, refines the relaxation based on properties of the discretizations, and does not involve a Lagrangian approach. Second, decremental state-space relaxations is a framework for improving column generation by incrementally strengthening a relaxed pricing problem (Righini and Salani 2008). This method uses column generation to solve its model which is not used in this work. It also does not maintain a static state-transition graph and only makes refinements that maintain an efficient labeling algorithm to solve the pricing problem. Third, iterative aggregation and disaggregation is a method that solves arc-flow formulations by creating an initial relaxed arc-flow formulation and strengthening it (Clautiaux et al. 2017). The work uses specific aggregations that are compiled differently than column elimination, requiring the graph to be rebuilt at each iteration, and does not incorporate a Lagrangian approach. Finally, a specialized branch-and-bound framework recently showed good computational results for solving arc-flow formulations (de Lima et al. 2022). It uses similar ideas to column elimination such as reduced-cost arc fixing, but it does not consider refining relaxations of the dynamic program in the same way as column elimination, nor does it consider a Lagrangian approach.

3. Problem Statement

Column elimination solves discrete optimization problems of a particular form. The form is a generalization of finding a minimum-cost sequence of elements from a finite set of feasible sequences, which appears, e.g., in discrete dynamic programming (Bellman 1957), domain independent dynamic programming (Kuroiwa and Beck 2024), and decision-diagram based optimization (Bergman et al. 2016). Here, the problem is to find a minimum cost *subset* of sequences of elements from a set of feasible sequences, where the set of feasible subsets can also be constrained. For our purposes, we allow the subset to contain multiple copies of the same sequence. We assume the cost of a subset of sequences is equal to the sum of the costs of individual sequences.

Formally, let U be a universe of elements and let $\mathcal{S}$ be a set of ordered sequences of elements in U, each with arbitrary but finite length. The discrete optimization problem is:

$$(\mathcal{P}) \quad \min_{X \subseteq \mathcal{S}} \left\{ \sum_{x \in X} f(x) : C(X) = 1 \right\} \tag{1}$$

where X represents the decision variable ranging over subsets of $\mathcal{S}$, function $C : 2^{\mathcal{S}} \to \{0,1\}$ defines feasible subsets (also considering multisets), and $f : \mathcal{S} \to \mathbb{R}$ is a cost function over $\mathcal{S}$. A main assumption of our model is that the function C can be represented as a conjunction of constraints with the following form. Each constraint is defined by a function $\gamma : \mathcal{S} \to \mathbb{R}$ that associates an additional 'cost' with each sequence, and has the form $\sum_{x \in X} \gamma(x) \leq b$. Denote $\Gamma = \{(\gamma_j, \circ_j, b_j)\}_{j=1}^m$ as the set of these constraints, where each $\circ_j$ is a comparator in the set $\{\geq, \leq, =\}$. We assume that the constraint function can be written as a conjunction of these new constraints, i.e. $C(X) = \wedge_{j=1}^m (\sum_{x \in X} \gamma_j(x) \circ_j b_j)$.

Many discrete optimization problems can be naturally described in this form; as a running example, we consider the *capacitated vehicle routing problem* (CVRP) (Toth and Vigo 2014).

Example 1. Let $\mathcal{G} = (V, A)$ be a complete directed graph with vertex set $V = \{0, 1, \ldots, n\}$ and arc set $A = \{(i,j) \mid i, j \in V, i \neq j\}$. Vertex 0 represents the depot, and vertices $\{1, \ldots, n\}$ represent the locations to be visited. We will interchangeably use vertices and locations. Each vertex $i \in V$ has a demand $q_i \geq 0$ and each arc $a \in A$ has a length $T_a \geq 0$ (typically representing time). Let K be the number of (homogeneous) vehicles, each with capacity Q. A *route* is a sequence of vertices $[v_1, v_2, \ldots, v_k]$ starting and ending at the depot with total demand at most Q. The *distance* of a route is the sum of its arc lengths, i.e., $\sum_{i=1}^{k-1} T_{(v_i, v_{i+1})}$. The CVRP consists in finding K routes such that each vertex except for the depot belongs to exactly one route and the sum of the route distances is minimized. We can describe the CVRP in the form of $\mathcal{P}$. Let $U = \{0, 1, \ldots, n\}$ and let $\mathcal{S}$ be the set of all (feasible) routes. The function f is a mapping from routes to their distances. The constraints C restrict the subset of routes to have cardinality K and to visit all locations, which can be translated to the following constraints in Γ. To ensure that each location is visited, we define a constraint for each $i \in \{1,\ldots,n\}$ by $(\gamma_i,=,1)$ where $\gamma_i(x) = [\![i \in x]\!]$, and $[\![\cdot]\!]$ denotes an indicator function. To ensure that each subset has K routes, we define a constraint by $(\gamma_{n+1},=,K)$ where $\gamma_{n+1}(x) = 1$ for all $x \in X$.

As a special case, observe that any integer linear programming problem can be represented in the form of $\mathcal{P}$. Consider the integer program $\min_{\eta \in \mathbb{Z}^n} \{\alpha^\mathsf{T}\eta : A\eta \geq \beta, \ell \leq \eta \leq u\}$, where $\alpha \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $\beta \in \mathbb{R}^m$, $\ell \in \mathbb{Z}^n$, and $u \in \mathbb{Z}^n$. We define the set of elements as the set of integers from the smallest lower bound to the largest upper bound, i.e. $U = \{\min_{i\in[n]} \ell_i, \min_{i\in[n]} \ell_i + 1, \ldots, \max_{i\in[n]} u_i\}$. We define $\mathcal{S} = \{[x_1,\ldots,x_n] : \ell_i \leq x_i \leq u_i, x_i \in \mathbb{Z}, \forall i \in \{1,\ldots,n\}\}$ for a fixed but arbitrary ordering of the variables x. In this case, each sequence in $\mathcal{S}$ is of the same length n. The cost function is $f(x) = \sum_{i=1}^n \alpha_i x_i$. The constraints C ensure that one sequence is chosen and that $A\eta \geq \beta$, which can be written in the form of Γ. To ensure that $A\eta \geq \beta$, we define constraints for each $j = 1,...,m$ by $(\gamma_j,\geq,\beta_j)$ where $\gamma_j(x) = \sum_{i=1}^n A_{ji}x_i$. To ensure one sequence is chosen, we define a constraint by $(\gamma_{m+1},=,1)$ where $\gamma_{m+1}(x) = 1$ for all $x \in X$. While this shows that column elimination can, in principle, be applied to integer programs of this general form, we do not expect this to be efficient unless we can exploit specific problem structures when defining problem $\mathcal{P}$. This will be discussed in the next section.

4. Modeling

Column elimination solves $\mathcal{P}$ via a particular modeling framework that combines dynamic programming and integer programming. We use a dynamic program to represent $\mathcal{S}$, f, and the cost functions in Γ. Then, an integer linear program is defined over the state-transition graph of the dynamic program to find the minimum cost subset of feasible sequences. The model resembles a common decomposition of the problem into an integer linear programming master problem and a dynamic programming pricing problem, used in column generation. In this section, we detail the model and describe relaxations of the model that are used in column elimination.

4.1. Dynamic Program

The set of sequences of elements $\mathcal{S}$, the cost function f, and the cost functions in Γ are modeled with dynamic programming. It is always possible to construct a dynamic program that encodes $\mathcal{S}$ with an arbitrary value function over the sequences (Hooker 2013). This can directly be extended to multiple value functions, in our case f and the cost functions in Γ. An advantage of dynamic programming is its ability to *compactly* represent the set of sequences (with the associated costs) instead of enumerating each one individually. We describe the structure of the dynamic program and explain how it models the set of sequences and costs.

We define a *dynamic program* by a set of states S, an initial state $r \in S$, a terminal state $t \in S$, a state transition function $h:(S \times U) \rightarrow S$, a cost function $c:(S \times U) \rightarrow \mathbb{R}$. A *solution* is a sequence of transitions $[(s_1,u_1),\ldots,(s_k,u_k)]$ where $(s_i,u_i) \in S \times U$ for $1 \leq i \leq k$, such that $s_1 = r$, $h(s_i,u_i) = s_{i+1}$ for $1 \leq i < k$, and $h(s_k,u_k) = t$. Each solution uses a unique set of elements $u = [u_1,...,u_k]$. The *cost* of a solution is $\sum_{i=1}^{k} c((s_i,u_i))$. We extend this standard definition by introducing a set of additional cost functions $G = \{g_j\}_{j=1}^{m}$ such that $g_j : (S \times U) \rightarrow \mathbb{R}$ for all $j = 1,...,m$. We denote the dynamic program as $P = (S,h,c,G)$, and assume that r and t are defined within S. We assume that a dynamic program is acyclic, so solutions cannot visit a state more than once.

We model $\mathcal{S}$, f, and the cost functions in Γ by constructing a dynamic program $P = (S,h,c,G)$ with the following properties. First, the set of solutions to P must be equal to $\mathcal{S}$. Second, for each solution $x \in \mathcal{S}$ equivalent to a solution $[(s_1,u_1),\ldots,(s_k,u_k)]$ in P, we require that $f(x) = \sum_{i=1}^{k} c((s_i,u_i))$ and $\gamma_j(x) = \sum_{i=1}^{k} g_j((s_i,u_i))$ for each $j = 1,...,m$. As mentioned above, it is always possible to construct a dynamic program with these properties (Hooker 2013).

Example 2. We formulate a dynamic program $P = (S,h,c,G)$ to represent $\mathcal{S}$, f, and the function costs in Γ for the problem $\mathcal{P}$ described for the CVRP in Example 1. Define each state in S by a tuple (NG, w, v), where NG is a ‘no-good’ set of visited locations, w is the current load, and v is the current location. The initial state r is $(\varnothing, 0, 0)$ and the terminal state t is a tuple of sentinel values (e.g. -1 to indicate a special state). The transition and cost functions are defined for two cases: visiting a location and returning to the depot. For each state $s = (\mathrm{NG}, w, v)$ and element $i \in U$ such that $i \notin \mathrm{NG}$, $i > 0$, and $w + q_i \leq Q$, define $h(s,i) = (\mathrm{NG} \cup \{i\}, w + q_i, i)$ and $c(s,i) = T_{(v,i)}$. For each state $s \in S$, define $h(s,0) = t$ and cost $c(s,0) = T_{(v,0)}$. For other combinations of states and decisions, we let the transition function and cost function equal -1 as a sentinel value. To model the cost functions in Γ, we define $g_j(s,i) = [\![i = j]\!]$ for each $j = 1,...,|V|$, and $g_{|V|+1}(s,i) = [\![s = r]\!]$.

4.2. Integer Linear Program

The optimization model that column elimination solves is an integer linear program defined over a network representation of P. The network representation is known as a *state-transition graph*, which is a directed acyclic graph where each state is represented by a node and each transition is represented by an arc. Given a dynamic program $P = (S, h, c, G)$, we define its state-transition graph as $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ with node set $\mathcal{N}$ and arc set $\mathcal{A}$. For each state $s \in S$ we introduce a node in $\mathcal{N}$. For each transition $h(s_i, u) = s_j$, we define an arc in $\mathcal{A}$ from the node for s_i to the node for s_j. Parallel arcs are distinguished by the transition element u. For ease of notation, especially for function inputs, we use nodes and states interchangeably, and we use arcs and state/element pairs interchangeably. This defines a one-to-one correspondence between sequences in $\mathcal{S}$ and directed r-t paths in $\mathcal{D}$.

The model is a constrained network flow problem over $\mathcal{D}$. For each arc $a \in \mathcal{A}$, we introduce a decision variable y_a. The model is as follows.

$$F: \quad \min \sum_{a \in \mathcal{A}} c(a) y_a \tag{2}$$

$$\text{s.t.} \sum_{a \in \mathcal{A}} g_j(a) y_a \ \circ_j \ b_j \qquad \forall\, j \in \{1, \ldots, m\} \tag{3}$$

$$\sum_{a \in \delta^+(s)} y_a - \sum_{a \in \delta^-(s)} y_a = 0 \qquad \forall\, s \in \mathcal{N} \setminus \{r, t\} \tag{4}$$

$$y_a \in \mathbb{Z}_+ \qquad \forall\, a \in \mathcal{A} \tag{5}$$

The objective (2) is to minimize the sum of the costs of the flows on arcs used in the solution. Constraints (3) are linear constraints using the additional cost functions from G with the comparators and right-hand side values from Γ. Constraints (4) are flow conservation constraints, where $\delta^+(s)$ and $\delta^-(s)$ are the sets of outgoing and incoming arcs respectively for a node $s \in \mathcal{N}$. Constraints (5) are nonnegativity and integrality constraints. We prove the correctness of the model in Theorem 1; an equivalent result is proven in Wolsey (1977).

THEOREM 1. *The optimal solution value of F is equal to the optimal solution value of $\mathcal{P}$.*

Proof We start by showing that the set of solutions to F is equal to the set of solutions to $\mathcal{P}$. Consider an optimal solution to F. The solution adheres to the flow conservation constraints (4) and integrality constraints (5), so by the flow decomposition theorem, the solution can be converted into a set of r-t paths (Ahuja et al. 1993). Each r-t path uses a set of arcs, equivalent to a sequence $x \in \mathcal{S}$. Let X be the set of these sequences (with multiplicity), and let A be the union of these sets of arcs (with multiplicity). The solution is feasible, so $\sum_{a\in A} g_j(a) \circ_j b_j$ for each $j = 1, ..., m$. By construction of the dynamic program, each $x \in \mathcal{S}$ corresponds to an arc set A' such that $\gamma_j(x) = \sum_{a\in A'} g_j(a)$ for each $j = 1, ..., m$. So, $\sum_{x\in X} \gamma_j(x) \circ_j b_j$. Thus, $C(X) = 1$. The reverse of this argument shows that $X \subseteq \mathcal{S}$ can be converted into a solution to F. To finish the proof, we show that each $X \subseteq \mathcal{S}$ has the same cost in F and $\mathcal{P}$. By construction of the dynamic program, each $x \in \mathcal{S}$ corresponds to an arc set A' such that $f(x) = \sum_{a\in A'} c(a)$. So, for a solution $X \subseteq \mathcal{S}$ corresponding to an arc set A' (both with multiplicity), $\sum_{x\in X} f(x) = \sum_{a\in A'} c(a)$. *Q.E.D.*

EXAMPLE 3. We give the complete formulation for the CVRP, which was originally developed in Karahalios and van Hoeve (2023) in online Appendix C.

4.3. Model Relaxations

Column elimination works by solving relaxations of F that are created by replacing P with relaxations of P. The literature contains two types of relaxations of dynamic programs. First, a *state-space relaxation* maps each state into a smaller state space such that predecessor states are conserved and each transition cost is the minimum cost of all equivalent transitions in the preimage of the mapping (Christofides et al. 1981). Second, in the decision diagram literature, relaxations are created by merging non-equivalent states and reasoning about the transitions to retain for the resulting state (Bergman et al. 2016). The key properties in both types of relaxations are that the relaxed dynamic program represents a superset of the sequences in the original dynamic program, and has a cost for each sequence that is less than or equal to the cost in the original dynamic program. While similar definitions have been proposed, we present a generalized notion of a dynamic program relaxation, without loss of generality for minimization problems, which is based on these properties:

Definition 1. Let P_1 and P_2 be dynamic programs with solution sets $\mathcal{S}_1$ and $\mathcal{S}_2$, solution costs that are defined by the functions f_1 and f_2, and additional costs defined by $\{\gamma_{1j}\}_{j=1}^m$ and $\{\gamma_{2j}\}_{j=1}^m$ with constraint operators $\{\circ_j\}_{j=1}^m$ and right-hand side values $\{b_j\}_{j=1}^m$. P_2 is a *dynamic program relaxation* w.r.t. P_1 if $\mathcal{S}_1 \subseteq \mathcal{S}_2$, $f_1(x) \geq f_2(x)$ for all $x \in \mathcal{S}_1$, and $\gamma_{2j}(x) \circ_j \gamma_{1j}(x)$ for all $j = 1, .., m$ and for all $x \in \mathcal{S}_1$.

State-space relaxations and relaxed decision diagrams both yield relaxed dynamic programs. We show in Appendix A how Definition 1 differs from the definition of a state-space relaxation.

We use a dynamic program relaxation to create a relaxation for the exact model F. Let $P' = (S', h', c', G')$ be a dynamic program relaxation w.r.t. P. Let $\mathcal{S}'$ be the set of solutions in P', let f' be the cost function represented by P', and let $\{\gamma'_j\}_{j=1}^m$ be the additional cost functions. We define F' as the model (2)-(5) based on P'. Theorem 2 shows that F' is a relaxation of F.

Theorem 2. *F' is a relaxation of F.*

Proof Because P' is a dynamic program relaxation w.r.t. P, the set of solutions (sequences) to P' is a superset of the set of solutions to P, i.e. $\mathcal{S} \subseteq \mathcal{S}'$. Recall from the previous proof that a solution to F (or F') can be mapped to a set of sequences $X \subseteq \mathcal{S}$ (or $X \subseteq \mathcal{S}'$). So, for each $X \subseteq \mathcal{S}$ that is feasible to F, X is feasible to F' because $X \subseteq \mathcal{S} \subseteq \mathcal{S}'$ and $\sum_{x \in X} \gamma'_j(x) \circ_j \sum_{x \in X} \gamma_j(x) \circ_j b_j$ for all $j = 1, ..., m$, and the objective function value is relaxed, i.e. $\sum_{x \in X} f(x) \geq \sum_{x \in X} f'(x)$. *Q.E.D.*

In practice, it is convenient when the functions in G only rely on the transition element from the input, and not the state, except the root state r. Then, we can use $G' = G$ in a dynamic program relaxation. This is the case for the covering constraints in our running example.

Example 4. Continuing our example on the CVRP, we describe a relaxation of the exact model F that is based on P. To do this, we formulate a dynamic program relaxation $P' = (S', h', c', G')$ w.r.t. P, using the well-known *ng*-route state-space relaxation (Baldacci et al. 2011b). An *ng*-route relaxation is defined by a parameter $\chi \in \mathbb{Z}$ and a set $N_i \subseteq V$ of size χ for each $i \in V$. It has the same state definition and cost function as P, but a different transition function h' which is again defined for two cases: visiting a location and returning to the depot. Given a state $s = (\mathrm{NG}, w, v)$ and element

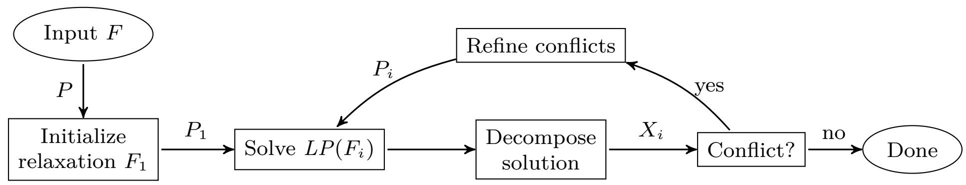


Figure 1 **Column elimination for solving** $LP(F)$**.**

$i \in U$ such that $i > 0$, $i \notin \mathrm{NG}$, and $w + q_i \leq Q$, let $h'(s,i) = ((\mathrm{NG} \cup \{i\}) \cap N_i, w + q_i, i)$. For each state $s \in S$, let $h'(s,0) = t$. For other combinations of states and decisions, we let the transition function equal -1 as a sentinel value. Because the constraints in the set G are defined only by the transition elements and r in P, we can keep the same constraints in G'.

5. Column Elimination

In this section, we describe column elimination for solving F. The main idea of column elimination is to start with an initial relaxation of F, and to iteratively strengthen the relaxation by updating the underlying dynamic program relaxation. The algorithm works by first solving the linear program relaxation of F, which we denote $LP(F)$, and then solving F. The purpose of solving $LP(F)$ is to efficiently strengthen the model relaxation and to achieve useful bounds, before trying to solve integer programs. At iteration i, we denote the model relaxation as F_i, which is created by P_i that represents the set of sequences $\mathcal{S}_i$. Below, we describe each step in more detail.

5.1. Solving the linear programming relaxation $LP(F)$

Column elimination solves $LP(F)$ by solving the linear programming relaxations of a series of improving relaxations, as shown in Figure 1. We next detail each of the steps of the algorithm.

Initializing a Relaxation: Given a model F, the first step is to create an initial relaxation F_1. This is done by defining a dynamic program relaxation w.r.t. P, denoted as P_1. The choice of initial relaxation can affect the performance of the algorithm, as there is often a tradeoff between the strength of a dual bound generated by F_1 and the number of variables in F_1, which affects the time required to achieve the dual bound.

Solving $LP(F_i)$: After setting up an initial relaxation, column elimination enters a loop of solving $LP(F_i)$ at each iteration i, decomposing the solution into a set of paths, and updating the dynamic program relaxation to remove any of these paths that are infeasible which we refer to as refining 'conflicts'. To solve $LP(F_i)$, column elimination can use an off-the-shelf linear programming solver or a subgradient method to solve an equivalent Lagrangian reformulation, which we describe in Section 6.

Path Decomposition: The solution to $LP(F_i)$ is decomposed into a set of sequences $X_i \subseteq \mathcal{S}_i$. Because F_i is a (constrained) network flow on a directed acyclic graph, such a path decomposition is known to exist (Ahuja et al. 1993), although it may not be unique for a given solution. If there exists a sequence $x \in X_i$ such that $x \notin \mathcal{S}$, then x has a 'conflict'. Otherwise X_i is an optimal solution for $LP(F)$. A conflict can also be due to a sequence having a relaxed cost in F_i, but for simplicity we will only consider a conflict as an infeasible sequence.

Conflict Refinement: The conflict refinement algorithm removes a conflict from P_i to create a new dynamic program relaxation w.r.t. P. Because the path decomposition may return multiple paths with conflicts, all of these can be used to refine the dynamic program relaxation. In our experimental results, however, we only remove one conflict at each iteration. We introduce Algorithm 1 as a general conflict refinement algorithm and prove its correctness. Similar refinement algorithms exist in the literature on decision diagrams (Hadzic et al. 2008, Ciré and Hooker 2014). There are more efficient problem-specific conflict refinement algorithms, such as the one for graph coloring in van Hoeve (2022), which uses terminology from decision diagrams. The description of the general algorithm is written in terms of updating the *dynamic program* (relaxation).

We describe Algorithm 1 as follows. Line 1 creates P_{i+1} as a copy of P_i. Line 2 sets a 'current state' s_{curr} as the root state $r \in S_{i+1}$. Line 3 begins a loop that iterates over the indices of elements in the conflict x. Lines 4 to 8 check the set of subsequences from r to s_{curr} in P' to see if any of these subsequences are feasible in P when appended with the next element x_j in the conflict. If none are feasible, then the transition from s_{curr} with x_j can be removed without removing any feasible sequences in $\mathcal{S}$, but it does remove the conflict x. In detail, Line 4 creates a set of states in P that are found by taking any possible transition from r to s_{curr} in P', where $S_{i+1}^{-s_{curr}}$ represents the set of these transitions and $P[y]$ represents the state in P found by starting from r and taking the transitions in y. Line 5 finds the set of feasible decisions from any of those states. Line 6 checks if the next element in the sequence x_j is not in the set of feasible decisions. Line 7 filters out the transition and Line 8 returns the updated dynamic program relaxation. Otherwise, Lines 10 to 18 create a new state which copies all of the information from the next state (found by the transition of s_{curr} with element x_j), which maintains all of the postsequences from the next state to t, but only $[x_1, ..., x_j]$ as a presequence from r to the new state. The uniqueness of the solution in $\mathcal{S}_i$ ensures that by the end of the algorithm the conflict x is removed. In detail, Line 10 finds the next state by taking the transition using s_{curr} and x_j. Line 11 creates a new state. Lines 12 to 16 copy the transitions and costs from the next state to this new state. Line 17 changes the transition from s_{curr} to the next state, to the newly created state. Line 18 updates s_{curr} to the new state.

Each step of the algorithm is straightforward to perform, and the step requiring the most computation is in Line 4 which requires considering all paths in a directed acyclic graph from the root to a node. In practice, this computation can be avoided by using information stored in each state to directly check the condition in Line 6. In our experiments, we use problem-specific conflict refinement algorithms that avoid creating a new node and new transitions in Lines 10 to 16 by instead using an existing node and transitions. This is done by relying on the structure of the relaxation in terms of its states, transitions, and costs to ensure that a dynamic program relaxation is maintained. We note two possible disadvantages of using existing nodes. First, updating a transition to equal an existing node may introduce infeasible sequences in P_{i+1} that were not solutions to P_i, because it combines presequences from the root that use the new transition to reach the existing node with the set of postsequences from the existing node to the terminal. Second, updating a transition to equal an existing node may introduce a cycle into the dynamic program for a similar. In our experiments, we find that the advantages of maintaining a smaller network flow formulation outweigh the potential disadvantages. We prove the correctness of Algorithm 1 as Theorem 3.

THEOREM 3. *Algorithm 1 outputs a dynamic program relaxation w.r.t. P, called P_{i+1}, such that P_i is a dynamic program relaxation w.r.t. P_{i+1} and $x \notin \mathcal{S}_{i+1}$.*

Algorithm 1 Conflict Refinement Algorithm

Input: A dynamic program $P = (S, h, c, G)$, a dynamic program relaxation $P_i = (S_i, h_i, c_i, G_i)$ with initial state r, and a conflict $x = [x_1, \ldots, x_k] \in \mathcal{S}_i$.

Output: P_{i+1}

1: $P_{i+1} := (S_{i+1}, h_{i+1}, c_{i+1}, G_{i+1}) = (S_i, h_i, c_i, G_i)$
2: $s_{curr} = r$
3: **for** $j = 1, \ldots, k$ **do**
4: $\quad S^- = \bigcup_{y \in \mathcal{S}_{i+1}^{-s_{curr}}} \{P[y]\}$
5: $\quad U_{S^-} = \bigcup_{s \in S^-} \{u : h(s,u) \neq -1\}$
6: $\quad$ **if** $x_j \notin U_{S^-}$ **then**
7: $\quad\quad h_{i+1}(s_{curr}, x_j) = -1$
8: $\quad\quad$ **return** P_{i+1}
9: $\quad$ **else**
10: $\quad\quad s_{next} = h_{i+1}(s_{curr}, x_j)$
11: $\quad\quad s_{new} = createState()$
12: $\quad\quad$ **for all** $u \in U$ **do**
13: $\quad\quad\quad h_{i+1}(s_{new}, u) = h_{i+1}(s_{next}, u)$
14: $\quad\quad\quad c_{i+1}(s_{new}, u) = c_{i+1}(s_{next}, u)$
15: $\quad\quad\quad$ **for all** $g_j \in G_{i+1}$ **do**
16: $\quad\quad\quad\quad g_j(s_{new}, u) = g_j(s_{next}, u)$
17: $\quad\quad h_{i+1}(s_{curr}, x_j) = s_{new}$
18: $\quad\quad s_{curr} = s_{new}$

Proof First, we show that P_{i+1} is a dynamic program relaxation w.r.t. P and that P_i is a dynamic program relaxation w.r.t. P_{i+1}. To start, P_{i+1} begins as a copy of P_i. The algorithm only updates P_{i+1} in Line 7 and Lines 10 to 17. In Line 7, a transition is made infeasible, which can only remove solutions from P_{i+1}, and only does so if the condition on Line 6 is met, which is equivalent to checking that no feasible solutions in P are removed. In Lines 10 to 16, a new state is created that is a copy of the next state found by starting at s_{curr} and transitioning with element x_j. Because the transition function outputs, costs, and additional costs are all copied, when Line 17 updates the transition from s_{curr} with x_j to be this new node, no solutions or solution costs change in P_{i+1}. So, the only changes from the solutions to P_i are that sequences not in $\mathcal{S}$ can be removed from P_{i+1}, which proves that

P_{i+1} is a dynamic program relaxation w.r.t. P and that P_i is a dynamic program relaxation w.r.t. P_{i+1}. Next, we prove that $x \notin \mathcal{S}_{i+1}$. The key observation is that at each step of the algorithm, P_{i+1} has exactly one presequence starting from r and transitioning to s_{curr}. In the first iteration this is trivially true. After that, s_{curr} is always updated to s_{new}, which is a new state that is only reachable by the previous s_{curr}. Thus, in the worst case, by the final iteration the only presequence to s_{curr} is $[x_1, ..., x_{k-1}]$ and then the transition with element x_k is infeasible in $\mathcal{S}$, so it will be removed in Line 7. Q.E.D.

To conclude this subsection, we prove the correctness of the column elimination algorithm for solving the linear programming relaxation of model F. Assume that the initial dynamic program relaxation has a cost function f' and additional cost functions $\{\gamma_j'\}_{j=1}^m$ such that for each $x \in \mathcal{S}$, $f'(x) = f(x)$ and $\gamma_j'(x) = \gamma_j(x)$ for each $j = 1, ..., m$.

Theorem 4. *Column elimination solves LP(F) in a finite number of steps.*

Proof Column elimination begins with a valid relaxation F_1. At each iteration i, column elimination solves $LP(F_i)$, and if there is a conflict, it is removed with Algorithm 1. By Theorem 3, removing a conflict from P_i creates a new dynamic program relaxation w.r.t P, denoted P_{i+1}, such that $\mathcal{S} \subseteq \mathcal{S}_{i+1} \subseteq \mathcal{S}_i$. There can only be a finite number of conflict refinements before $\mathcal{S}_i = \mathcal{S}$, because $\mathcal{S}_1$ and $\mathcal{S}$ are both finite and at least one solution is removed from $\mathcal{S}_i$ by Algorithm 1. Also, at each iteration of column elimination, each sequence $x \in \mathcal{S}$ will have costs $f(x)$ and $g_j(x)$ for all $j = 1, ..., m$, because Algorithm 1 does not update the costs of these sequences and the assumption that this holds for the initial dynamic program relaxation. So, if a solution to $LP(F_i)$ has no conflicts, then is also an optimal solution to $LP(F)$. *Q.E.D.*

5.2. Solving the integer programming model F

We describe three ways of solving the integer programming model F using column elimination. The first is a pure column elimination approach that extends column elimination for solving $LP(F)$ by iteratively solving integer programs F_i (at iteration i) with an off-the-shelf integer programming solver. The second approach, cut-and-refine, augments the approach for solving $LP(F)$ with cutting planes. The third approach, branch-and-refine, embeds the linear programming relaxation $LP(F)$ within a branch-and-bound search.

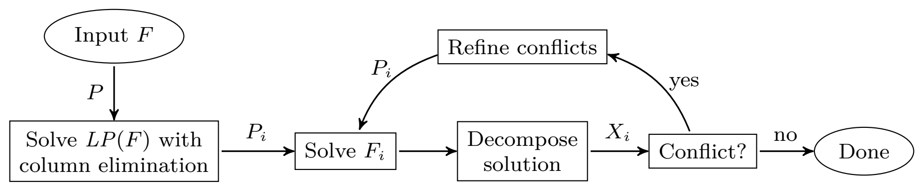


Figure 2 **Pure column elimination for solving F.**

5.2.1. Pure column elimination The pure column elimination approach extends column elimination for solving $LP(F)$ by continuing the iterative procedure, but at each iteration i solving F_i instead of $LP(F_i)$. The first step is to solve $LP(F)$, which strengthens the initial relaxation. Then, the algorithm iteratively solves strengthened relaxations of F as integer programs. The framework is shown in Figure 2.

To solve F_i, column elimination uses an off-the-shelf integer programming solver. This foregoes the need to develop problem-specific cuts or branching rules, although we show how to incorporate these in the next section. Conflicts are identified and refined in the same way as for solving $LP(F)$.

There are two advantages to solving $LP(F)$ before solving F. First, a solution to $LP(F_i)$ may contain conflicts that also appear in an optimal solution to F_i, but $LP(F_i)$ is easier to solve. Second, a solution to $LP(F_i)$ provides a lower bound, which is likely weaker than the optimal solution value to the integer relaxation F_i, but again it is easier to obtain. Linear programming lower bounds are useful to reduce the size of the problem by a method called variable fixing, which we describe in Appendix D. We show that column elimination solves F in Theorem 5.

THEOREM 5. *Column elimination solves F in a finite number of steps.*

Proof Column elimination solves $LP(F)$ in a finite number of steps by Theorem 4. Then, by the same logic as the proof of Theorem 4, only a finite number of conflicts can be refined before $\mathcal{S}_i = \mathcal{S}$. So, when there are no conflicts, an optimal solution to F_i is also an optimal solution to F. *Q.E.D.*

5.2.2. Cut-and-refine Cut-and-refine introduces cutting planes to the column elimination algorithm by not only refining conflicts in solutions to $LP(F_i)$ but also adding valid inequalities to remove conflict-free solutions. The framework is depicted in Figure 3. When an optimal solution to $LP(F_i)$

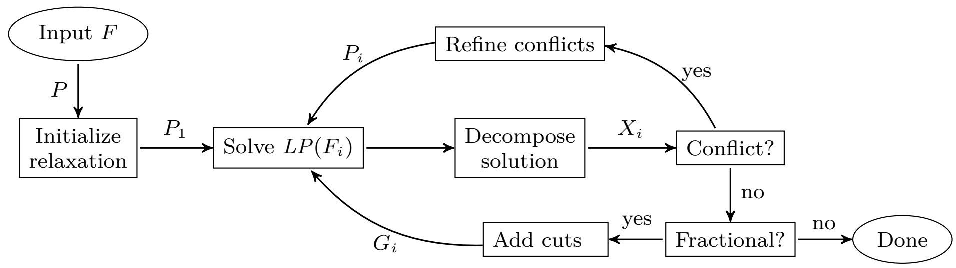


Figure 3 **Cut-and-refine for solving F.**

is fractional, cut-and-refine adds one or more valid cuts to remove that solution. Each cut should have a form that can be represented even if the underlying dynamic program relaxation is changed in a future iteration. Cut-and-refine, although not yet given this name, was shown to improve the performance of column elimination on some instances of the CVRP in (Karahalios and van Hoeve 2023), which used problem-specific cuts. We note that cuts deemed 'robust' in the column generation literature were effectively implemented in that work, but non-robust cuts were not; for more discussion see Appendix J. We prove the correctness and finite termination of cut-and-refine in Corollary 1, with the assumption that cuts are added from a finite cutting plane procedure.

COROLLARY 1. *Cut-and-refine solves F in a finite number of steps.*

Proof A finite number of refinements are needed to obtain F from F_1. When a solution to $LP(F_i)$ does not contain a conflict, only a finite number of cuts need to be added before the solution either contains a conflict or is integer-valued, because of the finite cutting plane procedure. *Q.E.D.*

5.2.3. Branch-and-refine Branch-and-refine embeds column elimination in a branch-and-bound framework. The algorithm begins by solving $LP(F)$ with column elimination at the root node, and then proceeds with branch-and-bound. The branching rule must partition the set of solutions in a way that maintains the form of the problem as $\mathcal{P}$, so column elimination can solve the subproblems.

It may not be straightforward to create such a branching rule. For example, branching on a single variable y_a in a formulation F_i may be complicated after a conflict refinement, because the arc a needs to be mapped from P_i to P_{i+1}. Similar considerations apply to branch-and-price algorithms, in which branching decisions often depend on problem-specific features. For example, for the CVRP a constraint can be imposed on the number of times that a set of routes enters and exits a set of locations, which can either be at most twice or at least four times, and can be expressed in the underlying dynamic program formulation. In the online Appendix K we provide an experimental study of solving the vertex coloring problem using branch-and-refine. We prove both the correctness and finite termination of branch-and-refine in Corollary 2.

COROLLARY 2. *Branch-and-refine solves F in a finite number of steps.*

Proof Because the set of feasible subsets in $\mathcal{P}$ is a finite set, any branch-and-bound tree will have a finite number of nodes. Column elimination can solve each subproblem in a finite number of steps. *Q.E.D.*

A natural extension is to embed a cutting plane procedure from Section 5.2.2 into the branch-and-bound search to strengthen the linear programming relaxations. This would yield a branch-cut-and-refine algorithm.

6. Column Elimination with Subgradient Descent

In this section, we propose solving the linear programming relaxation $LP(F)$ via a Lagrangian reformulation with subgradient descent. A key benefit of this approach is the ability to refine conflicts while solving the linear programming relaxation, instead of requiring an optimal solution before refinement. This can be particularly helpful for large-scale instances for which solving the linear program relaxations can be a computational bottleneck. We present a generalization of the single-path Lagrangian approach by Tang and van Hoeve (2024), which has a simpler subproblem that is a single path through the arc-flow model, and the application-specific approach by Karahalios and van Hoeve (2023), which was presented specifically for the capacitated vehicle routing problem.

6.1. Lagrangian Model

Recall that the linear programming relaxation $LP(F)$ contains constraints (3). Without loss of generality, we assume in this section that these constraints are of the following standard form:

$$\sum_{a \in \mathcal{A}} g_j(a) y_a \geq b_j \qquad \forall j \in \{1, \ldots, m\}.$$

We create a Lagrangian formulation for $LP(F)$ by introducing a Lagrangian dual multiplier $\lambda_j \geq 0$ for each $j \in \{1,\ldots,m\}$ and defining the Lagrangian relaxation as follows:

$$L(\lambda): \quad \min \sum_{a\in\mathcal{A}} c_a y_a + \sum_{j=1}^{m} \lambda_j (b_j - \sum_{a\in\mathcal{A}} g_j(a) y_a) \tag{6}$$

$$\text{s.t.} \sum_{a\in\delta^-(u)} y_a - \sum_{a\in\delta^+(u)} y_a = 0 \qquad \forall u \in \mathcal{N}\setminus\{r,t\} \tag{7}$$

$$y_a \geq 0, y_a \in \mathbb{Z} \qquad \forall a \in \mathcal{A} \tag{8}$$

The objective function (6) can be rewritten as

$$\min \sum_{a\in\mathcal{A}} c_a y_a - \sum_{j=1}^{m} \lambda_j \sum_{a\in\mathcal{A}} g_j(a) y_a + \sum_{j=1}^{m} \lambda_j b_j =$$

$$\min \sum_{a\in\mathcal{A}} (c_a - \sum_{j=1}^{m} g_j(a)\lambda_j) y_a + \sum_{j=1}^{m} \lambda_j b_j.$$

Each relaxation $L(\lambda)$ corresponds to solving a (continuous) minimum-cost network flow problem on a directed acyclic graph. The Lagrangian formulation is $\max_{\lambda\geq 0} L(\lambda)$, which has an optimal solution value equal to the optimal solution value of $LP(F)$ (Geoffrion 1974). It can be solved by a subgradient descent method (Nemhauser and Wolsey 1988). For fixed λ, the Lagrangian relaxation $L(\lambda)$ can be solved efficiently using a successive shortest paths algorithm (Ahuja et al. 1993). As a special case, when the problem has at most unit flow on the arcs, a 'minimum update' successive shortest paths algorithm can solve this problem more efficiently in practice (Wang et al. 2019). We discuss the computational benefits of using the Lagrangian relaxation in comparison to the standard linear programming relaxation, and the minimum update algorithm in comparison with a standard successive shortest paths algorithm in the online Appendix F.

Given an upper bound K on the number of r-t paths in an optimal solution to the Lagrangian relaxation for a fixed λ, the following proposition gives an upper bound on the runtime. A proof is given in Karahalios and van Hoeve (2023) for the CVRP, which also applies to the general case.

Proposition 1. *For a fixed λ, $L(\lambda)$ can be solved in $O(K(|\mathcal{N}|\log(|\mathcal{N}|)+|\mathcal{A}|))$ time.*

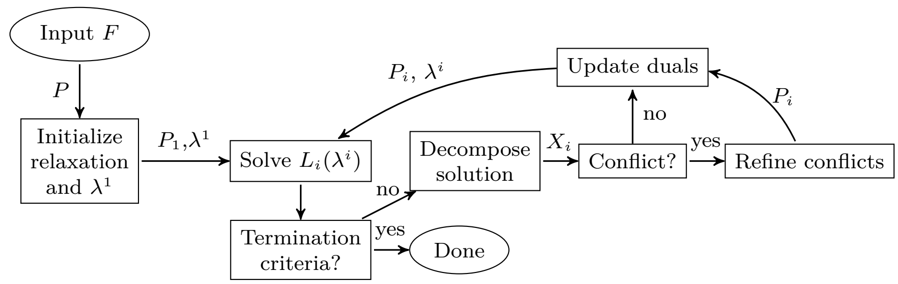


Figure 4 **Column elimination for solving $LP(F)$ using subgradient descent.**

6.2. Subgradient Descent

We next modify column elimination to incorporate solving $LP(F)$ with subgradient descent. At iteration i, we denote the Lagrangian relaxation as $L_i(\lambda)$ where λ are the dual variables. The updated algorithm is given in Figure 4. While the column elimination framework presented in Figure 1 solves $LP(F_i)$ at each iteration i, and then refines conflicts based on the optimal solution, the framework in Figure 4 instead solves $L_i(\lambda^i)$ at each iteration, where λ^i is the value of λ at iteration i. This allows the algorithm to use the resulting solution to the Lagrangian relaxation to both update the dynamic program relaxation and take a step to update the dual values. We discuss the convergence of the algorithm in Appendix E.

There are two possible advantages to using column elimination with subgradient descent. First, the dynamic program P_i can be refined more quickly by not needing the optimal solution to $LP(F_i)$ before refining conflicts. The refinements can still be effective in removing the optimal solution to $LP(F_i)$, because an average of the subproblem solutions converges to an optimal solution (Anstreicher and Wolsey 2009). Second, column elimination can apply variable fixing at each iteration of subgradient descent, which can accelerate the method. It is useful that subgradient descent can try variable fixing at each step, because the result of variable fixing can vary depending on the feasible dual solution used. Moreover, suboptimal dual solutions often produce a greater effectiveness in variable fixing because they can remove arcs with a positive primal value in the current optimal basis (de Lima et al. 2022). Subgradient descent does not require the dual values at each step to be feasible, so in these cases the dual values can be 'repaired' to a nearby feasible solution. In our experiments, we use a repair algorithm that finds the most violated constraint and updates one dual value at a time until the constraint is satisfied.

Two variations of column elimination with subgradient descent are given by Tang and van Hoeve (2024): 'LagAdapt' and 'LagRestart'. LagAdapt stops updating the duals after some stopping criteria, but continues solving the Lagrangian relaxation and refining conflicts. LagRestart updates the duals until some number of conflicts are found, then refines all of the conflicts, and resets the subgradient descent algorithm, including the duals and iteration count, which affects the step size. We considered these variants in our experiments, but did not see improvements.

6.3. Cut-and-refine with Subgradient Descent

Modifying cut-and-refine to incorporate solving $LP(F)$ with subgradient descent is not straightforward. Indeed, a previous work Lucena (2005) discusses the difficulty of effectively adding cuts during subgradient descent, within their framework called relax and cut. A relax and cut procedure can either be 'delayed', meaning cuts are only identified when subgradient descent terminates, or 'non-delayed', meaning cuts are added during subgradient descent. Once cuts are identified, they are added by 'dualizing' the cut and starting subgradient descent with the new variables. Using the delayed approach, Karahalios and van Hoeve (2023) added a limited number of cuts to column elimination with subgradient descent. The experiments from their work show a performance improvement for solving capacitated vehicle routing problems. We also use the delayed approach in the experiments in this work. Having mentioned relax-and-cut, it is worth noting that column elimination with subgradient descent – without adding cutting planes to the formulation – is similar to relax-and-cut where the 'cuts' are refinements to the dynamic program relaxation.

7. Applications

In addition to the CVRP, we will evaluate the computational performance of column elimination on four other fundamental combinatorial optimization problems: the vehicle routing problem with time windows, the graph multicoloring problem, the pickup and delivery problem with time windows, and the sequential ordering problem. We present here the problem definitions, while the associated dynamic programming models and initial dynamic program relaxations can be found in Appendix B.

VRPTW The *vehicle routing problem with time windows* (VRPTW) is a generalization of the CVRP that introduces constraints that each location must be visited in a given time window (Toth and Vigo 2014). We modify the definition of the CVRP from Example 1 and introduce for each location $i \in V$ a time window $[e_i, l_i]$. The definition of a *route* is updated to be a sequence of vertices $[v_1, v_2, \ldots, v_k]$ starting and ending at the depot with total demand at most Q, such that each location is visited during its time window. The requirement to use exactly K vehicles is relaxed. A vehicle is allowed to wait at a location until the start of the location's time window.

Graph Multicoloring We define the *graph multicoloring problem* as follows (Gualandi and Malucelli 2012). Given an undirected graph $\mathcal{G} = (V, E)$ and weights $b_v \in \mathbb{Z}$ for each $v \in V$, use the minimum number of colors possible to color each vertex v with b_v colors such that adjacent vertices are not assigned any of the same colors. Define a subset of vertices that are pairwise non-adjacent as an *independent set*. So, for each color, the set of vertices assigned that color must be an independent set. The *vertex coloring problem* is the graph multicoloring problem with $b_v = 1$ for all $v \in V$.

PDPTW The *pickup and delivery problem with time windows* (PDPTW) is a generalization of the VRPTW, with the addition of precedence constraints (Ropke et al. 2007). Specifically, this is the '1-1 PD' problem where each demand has a specific origin and a specific destination, in contrast to other variants (Berbeglia et al. 2010). The notation is the same as for the VRPTW, but now the locations are partitioned into origin-destination pairs. The locations are labeled $V = \{0, 1, \ldots, 2n\}$ and partitioned into the depot node 0, and sets $\Phi = \{1, \ldots, n\}$ and $\Omega = \{n+1, \ldots, 2n\}$ which represent pickup and delivery nodes respectively. For each origin-destination pair, the demand of the destination is negative the demand of the origin. So, for each $i \in \Phi$, $q_i = -q_{i+n}$. A *route* has the same requirements as for the VRPTW, but now also includes precedence constraints that an origin node $i \in \Phi$ must be visited before its corresponding delivery node $i + n \in \Omega$ by the same vehicle. The problem is to minimize the total distance traveled by a set of feasible routes that visit all of the locations.

SOP The *sequential ordering problem* corresponds to the precedence-constrained (asymmetric) traveling salesman problem. It is a special case of the PDPTW on a single vehicle, without time windows or vehicle capacity and for which precedences are defined for a subset of pairs of locations.

8. Experimental Results

In this section, we provide an experimental evaluation of the computational performance of column elimination. We first consider the impact of the various components of the column elimination algorithm, and then give a comparison with the state-of-the-art.

8.1. Experimental Setup

All experiments are run on an Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz. We use CPLEX version 22.1 with one thread and default parameters. The best-known primal solution value is input to all solvers. For each experiment, we give each algorithm a timeout of 3,600 seconds. The Github repositories of the code will be made available upon acceptance of the paper.

Initial Relaxation: We use the following initial relaxations as defaults for each application, unless otherwise specified. For VRPTW, CVRP, PDPTW, we follow the description in Appendix B, starting with an initial *ng*-route relaxation with $\rho = 2$. For the VRPTW and PDPTW, we relax the capacity constraints for instances in the classes R2, C2, RC2 and we keep the capacity constraints for all others (Gehring and Homberger 2002). For VRPTW and PDPTW instances, we also set bucketing parameter Δ equal to the minimum non-zero service time, further described in Appendix B. For graph multicoloring and vertex coloring instances, we start with the initial relaxation that is described in Appendix B.

Subgradient Descent: We make the following implementation decisions for column elimination with subgradient descent. At each iteration k of the subgradient method, we use a subgradient γ^k such that for each $j = 1, ..., m$, $\gamma_j^k = (b_j - \sum_{a \in \mathcal{A}} g_j(a) y_a^k)$ where y_a^k is the solution to $L_k(\lambda^k)$. We use an estimated Polyak step size $\alpha^k = \frac{\psi^* - v(\lambda^k)}{||\gamma^k||_2^2}$ where $\psi^* = LB * (1 + \frac{5}{100+k})$ is an estimate of the optimal value, k is the iteration, LB is the best lower bound so far, and $v(\lambda)$ is the optimal value of $L_k(\lambda^k)$. To update the multipliers, we set $\lambda^{k+1} = \lambda^k + \alpha^k \gamma^k$ (Bertsekas 2009). For the VRPTW/CVRP, we use initial dual values $\lambda_i^1 = 2l_{(0,i)} \frac{q_i}{Q}$ for each $j = 1, ..., m$. For the SOP, we initialize λ_j^1 for each $j = 1, ..., m$ to the best known primal bound divided by the number of locations. For vertex coloring, we initialize $\lambda_j^1 = 0$ for each $j = 1, ..., m$. We do not 'dualize' constraints on the size of the subset of feasible sequences, like for the CVRP. We repair infeasible λ values using a greedy algorithm. We store the dual values that give the largest percent of fixed arcs, and we use these dual values for arc fixing at each iteration. We switch to using column elimination with CPLEX when variable fixing has reduced the number of arc variables to below 100,000 or by at least 97.5% of the total arcs.

Cutting Planes: For the VRPTW and CVRP, we give cut-and-refine the following defaults. We use the package CVRPSEP (Lysgaard 2003) to add at most 100 rounded capacity cuts at each iteration of column elimination. We do not add cuts during column elimination with subgradient descent.

8.2. Impact of Column Elimination Components

We performed an extensive evaluation of the various components of column elimination, using the different applications listed above for different purposes. We give details on the respective problem domains and the computational results in the online Appendices F-K. As a summary of these results, we report the following insights:

Subgradient Descent: Column elimination with subgradient descent performs well on VRPTW instances, but not on vertex coloring instances. This is likely due to the (non-)stability of the primal and dual solutions for successive iterations. (See Appendix F.)

Initial Relaxation: Larger initial relaxations are not always better. Our experimental study on the SOP shows that an initial relaxation that includes important constraints can perform well when the size of the state-transition graph is not too large. Otherwise, a weaker initial relaxation that corresponds to a smaller state-transition graph performs better. (See Appendix G.)

Minimum-Update SSP: The specialized minimum-update successive shortest paths algorithm is very effective, as it can solve the Lagrangian subproblems on average 3.7 times faster than a standard successive shortest paths algorithm for the CVRP. (See Appendix I.)

Variable Fixing: Variable fixing is critical to solving large-scale instances. Using variable fixing allows column elimination to solve CVRP instances on average 53% more quickly than without variable fixing. (See Appendix H.)

Cut-and-Refine: We show for the CVRP that cut-and-refine can be effective for column elimination using a linear programming solver, but is not as effective for column elimination with subgradient descent. This is due to the challenges related to integrating cutting planes into the Lagrangian reformulation and subgradient descent. (See Appendix J.)

Branch-and-Refine: Branch-and-refine can help when conflict refinement alone 'plateaus' at a sub-optimal bound. We find that it solves instances of the vertex coloring problem on average 2.3 times faster than without branching. (See Appendix K.)

8.3. Comparison with State-of-the-Art Methods

We next compare column elimination to state-of-the-art algorithms for solving the VRPTW, the graph multicoloring problem, and the PDPTW. Full details of these results can be found in the (online) appendix. For each instance, we list the current best-known upper bound (UB) that each method takes as input. We report for each method the lower bound (LB) and solution time (Time (s)) and, where applicable, the number of explored search nodes (Nodes). For column elimination, we also report the number of column elimination iterations (CEIt) and column elimination with subgradient descent iterations (CESIt), the number of refinements (CR), and the number of cuts (Cuts). Below we provide a summary of the computational findings for each application.

8.3.1. VRPTW We use column elimination to solve the VRPTW instances from Gehring and Homberger (2002). We compare the performances of column elimination and the state-of-the-art solver VRPSolver (Pessoa et al. 2020), which is based on branch-and-cut-and-price. We use VRP-Solver with the default settings on the same server as the one we use for column elimination. For a fair comparison, we also initialize VRPSolver with the best-known upper bounds.

Before discussing the results, we consider the characteristics of the six classes of instances in the benchmark set: C1,C2,R1,R2,RC1,RC2. For C1 and C2, the locations are generated in clusters. For R1 and R2, the locations are randomly generated. For RC1 and RC2, some locations are clustered and some are randomly generated. For R1, C1, and RC1, each feasible route has few customers due to a short time horizon, which effectively removes the capacity constraint. For R2, C2, and RC2, feasible routes can have many customers and the capacity Q is large. In fact, VRPSolver relaxes the capacity constraint for problems in instances R2, C2, and RC2. So, we do the same in our initial relaxations. We hypothesize that column elimination will perform better when two characteristics of an instance hold. First, a strong relaxation of the set of feasible routes can be compactly represented. Second, conflict refinement can quickly close the gap between the initial relaxation and the full model. Instances with clustered locations are likely to require eliminating routes that have cycles within the clusters, but can leave many routes relaxed that have cycles across more than one cluster. This may allow a strong relaxation to be obtained quickly and remain over a compact network. Similarly, instances for which we use an initial dynamic program relaxation that does not maintain capacity in its states allows for a more compact network, so capacity values are only needed when they are the reason a useful route in the solution to a relaxation is infeasible. These characteristics allow the conflict refinements to strengthen relaxations. Thus, we hypothesize that column elimination may work well on C2 instances.

The results in Table 1 compare the performance of column elimination and VRPSolver for C2 instances with 400 and 600 locations. The table shows that column elimination can outperform VRPSolver on five instances. Table 2 shows the same comparison for three instances that column elimination solves. Based on CVRPLib (`http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`), column elimination is able to close one instance and solve two others that were only recently closed by VRPSolver in work that is not yet published. We give a summary of the results in Table 3. We show the full table of results for all instance classes in Appendix L.

8.3.2. Graph Multicoloring We compare column elimination to the branch-and-price method from Gualandi and Malucelli (2012) (GM) for solving the COG graph multicoloring instances introduced in the same paper. We directly use the results from Gualandi and Malucelli (2012) from their paper, as their code is not available. The server used in their paper is not specified, but likely slower than ours. Also, to be fair, we do not initialize column elimination with the best-known bound. The full table of results are presented in Appendix M.

The results in Table 4 show that column elimination closes five instances by obtaining an optimal solution at termination: COG-gesa2-o, COG-misc07, COG-nsrand-ipx, COG-opt1217, and COG-rout. Column elimination solves four of these instances without making any conflict refinements.

We attribute this to the initial relaxation that column elimination uses, which is not used in the branch-and-price method by Gualandi and Malucelli (2012). The results suggests that smaller relaxed models should be considered for sparse graphs. For some summary statistics, GM solves 9 instances to optimality and fails to give a dual bound for 5 instances. Column elimination solves 10 instances to optimality and fails to give a dual bound on only 3 instances.

8.3.3. PDPTW We compare column elimination to a dual ascent method from Baldacci et al. (2011a) (BBM) and VRPSolver for solving instances from Li and Lim (2001). We directly take the results from Baldacci et al. (2011a) from their paper as their code is not available. We use VRPSolver with the default settings and use the same server as the one we use for column elimination. The full table of results are presented in Appendix N.

The results show that BBM outperforms column elimination on all instances, although column elimination finds competitive bounds for many instances. In contrast, the general VRPSolver does not find a lower bound for any instance. Because no exact method including BBM and VRPSolver has reported results on many of the larger instances from Li and Lim (2001), we show 14 of these instances in Table 5 that column elimination closes.

We give a full table of results in Appendix N. For high level statistics, when comparing only instances that have been reported on by a state-of-the-art exact solver, column elimination solves 8 instances, fails to provide a lower bound on 5 instances, and computes an average optimality gap of 15.6% on the remaining instances. In comparison, the state-of-the-art solver Baldacci et al. (2011a) solves 23 instances, fails to provide a lower bound on 0 instances, and computes an average optimality gap of 11.7% on the remaining instances. For the instances that have not been reported on by an exact solver, column elimination solves 14 instances, fails to provide a bound on 0 instances, and computes an average optimality gap of 33.6% on the remaining instances.

9. Conclusion

We introduced column elimination as an iterative framework for solving a general class of discrete optimization problems. The framework models these problems by a minimum-cost constrained network flow problem over the state-transition graph of a dynamic program that stores the feasible sequences, their costs, and additional costs. We generalized earlier work by introducing the concept of dynamic programming relaxations, and described the column elimination algorithm as an iterative procedure that solves increasingly stronger dynamic programming relaxations of the problem by removing infeasible solutions. This iterative method converges in a finite number of steps to the optimal solution. As a variant, we presented a subgradient method that uses a minimum update successive shortest paths algorithm to solve the Lagrangian relaxation of the network flow problem. We showed that this method can solve the Lagrangian method more efficiently, but also allows refining the dynamic program relaxations more quickly. We also introduced cut-and-refine and branch-and-refine as extensions of the algorithm. Lastly, we showed experimentally that column elimination can be especially effective for large-scale instances, closing five open instances of the graph multicoloring problem, one open instance with 1,000 locations of the vehicle routing problem with time windows, and 14 open instances of the pickup-and-delivery problem with time windows.

References

Ahuja RK, Magnanti TL, Orlin JB (1993) *Network flows* (Prentice Hall).

Anstreicher KM, Wolsey LA (2009) Two "well-known" properties of subgradient optimization. *Mathematical Programming* 120(1):213–220.

Appelgren LH (1971) Integer programming methods for a vessel scheduling problem. *Transportation Science* 5(1):64–78.

Ascheuer N, Jünger M, Reinelt G (2000) A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications* 17:61–84.

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

Baldacci R, Mingozzi A, Roberti R (2011b) New route relaxation and pricing strategies for the vehicle routing problem. *Operations research* 59(5):1269–1283.

Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MWP, Vance PH (1998) Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research* 46(3):316–329.

Bellman R (1957) *Dynamic Programming* (Princeton University Press).

Berbeglia G, Cordeau JF, Laporte G (2010) Dynamic pickup and delivery problems. *European journal of operational research* 202(1):8–15.

Bergman D, Ciré AA (2018) Discrete Nonlinear Optimization by State-Space Decompositions. *Management Science* 64(10):4700–4720.

Bergman D, Cire AA, Van Hoeve WJ, Hooker JN (2016) *Decision diagrams for optimization* (Springer).

Bertsekas D (2009) *Convex optimization theory*, volume 1 (Athena Scientific).

Boland N, Hewitt M, Marshall L, Savelsbergh M (2017) The continuous-time service network design problem. *Operations research* 65(5):1303–1321.

Bulhoes T, Sadykov R, Subramanian A, Uchoa E (2020) On the exact solution of a large class of parallel machine scheduling problems. *Journal of Scheduling* 23(4):411–429.

Castro MP, Ciré AA, Beck JC (2022) Decision Diagrams for Discrete Optimization: A Survey of Recent Advances. *INFORMS Journal on Computing* 34(4):2271–2295.

Chen ZL, Powell WB (1999) Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing* 11(1):78–94.

Christofides N, Mingozzi A, Toth P (1981) State-space relaxation procedures for the computation of bounds to routing problems. *Networks* 11(2):145–164.

Ciré AA, Hooker JN (2014) The Separation Problem for Binary Decision Diagrams. *Proceedings of ISAIM.*

Ciré AA, van Hoeve WJ (2013) Multivalued Decision Diagrams for Sequencing Problems. *Operations Research* 61(6):1411–1428.

Clautiaux F, Hanafi S, Macedo R, Voge ME, Alves C (2017) Iterative aggregation and disaggregation algorithm for pseudo-polynomial network flow models with side constraints. *European Journal of Operational Research* 258(2):467–477.

Corneil DG, Graham B (1973) An algorithm for determining the chromatic number of a graph. *SIAM Journal on Computing* 2(4):311–318.

Dantzig GB, Wolfe P (1960) Decomposition principle for linear programs. *Operations research* 8(1):101–111.

de Lima VL, Alves C, Clautiaux F, Iori M, de Carvalho JMV (2022) Arc flow formulations based on dynamic programming: Theoretical foundations and applications. *European Journal of Operational Research* 296(1):3–21.

Focacci F, Lodi A, Milano M (1999) Cost-based domain filtering. *Principles and Practice of Constraint Programming–CP'99: 5th International Conference, CP'99, Alexandria, VA, USA, October 11-14, 1999. Proceedings 5*, 189–203 (Springer).

Ford LR, Fulkerson DR (1958) A Suggested Computation for Maximal Multi-Commodity Network Flows. *Management Science* 5(1):97–101.

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

Gehring H, Homberger J (2002) Parallelization of a Two-Phase Metaheuristic for Routing Problems with Time Windows. *Journal of Heuristics* 8:251–276.

Geoffrion AM (1974) Lagrangian Relaxation for Integer Programming. *Mathematical Programming Study 2* 82–114.

Gouveia L, Leitner M, Ruthmair M (2019) Layered graph approaches for combinatorial optimization problems. *Computers & Operations Research* 102:22–38.

Gualandi S, Malucelli F (2012) Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing* 24(1):81–100.

Hadzic T, Hooker JN, O'Sullivan B, Tiedemann P (2008) Approximate compilation of constraints into multivalued decision diagrams. *International Conference on Principles and Practice of Constraint Programming*, 448–462 (Springer).

Held S, Cook W, Sewell EC (2011) Safe lower bounds for graph coloring. *International Conference on Integer Programming and Combinatorial Optimization*, 261–273 (Springer).

Held S, Cook W, Sewell EC (2012) Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation* 4(4):363–381.

Hoogendoorn YN, Dalmeijer K (2025) Resource-robust valid inequalities for vehicle routing and related problems. *Operations Research Letters* 107355.

Hooker JN (2013) Decision diagrams and dynamic programming. *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 94–110 (Springer).

Horn DIM (2021) *Advances in Search Techniques for Combinatorial Optimization: New Anytime A Search and Decision Diagram Based Approaches*. Ph.D. thesis, Technische Universität Wien.

Irnich S, Desaulniers G, Desrosiers J, Hadjar A (2010) Path-reduced costs for eliminating arcs in routing and scheduling. *INFORMS Journal on Computing* 22(2):297–313.

Johnson DS, Trick MA (1996) *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26 (American Mathematical Soc.).

Karahalios A, van Hoeve WJ (2022) Variable ordering for decision diagrams: A portfolio approach. *Constraints* 27(1):116–133.

Karahalios A, van Hoeve WJ (2023) Column elimination for capacitated vehicle routing problems. *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 35–51 (Springer).

Kowalczyk D, Leus R, Hojny C, Røpke S (2024) A flow-based formulation for parallel machine scheduling using decision diagrams. *INFORMS Journal on Computing* 36(6):1696–1714.

Kuroiwa R, Beck JC (2024) Domain-Independent Dynamic Programming. ArXiv preprint arXiv:2401.13883.

Leus R, Kowalczyk D (2016) Improving column generation methods or sheduling problems using zdd and stabilization. *2016 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 99–103 (IEEE).

Li H, Lim A (2001) A metaheuristic for the pickup and delivery problem with time windows. *Proceedings 13th IEEE International Conference on Tools with Artificial Intelligence. ICTAI 2001*, 160–167 (IEEE).

Libralesso L, Bouhassoun AM, Cambazard H, Jost V (2020) Tree Search for the Sequential Ordering Problem. *ECAI 2020 Frontiers in Artificial Intelligence and Applications* (Santiago de compostella, Spain), URL `http://dx.doi.org/10.3233/FAIA200126`.

Lozano L, Bergman D, Cire AA (2022) Constrained shortest-path reformulations for discrete bilevel and robust optimization. ArXiv preprint arXiv:2206.12962.

Lübbecke ME, Desrosiers J (2005) Selected topics in column generation. *Operations research* 53(6):1007–1023.

Lucena A (2005) Non delayed relax-and-cut algorithms. *Annals of Operations Research* 140(1):375–410.

Lysgaard J (2003) CVRPSEP: A package of separation routines for the capacitated vehicle routing problem. URL `https://github.com/sassoftware/cvrpsep`.

Mandal U, Regan A, Rousseau LM, Yarkony J (2023) Graph Master and Local Area Routes for Efficient Column Generation for the Capacitated Vehicle Routing Problem with Time Windows. ArXiv preprint arXiv:2304.11723.

Mehrotra A, Trick MA (1996) A column generation approach for graph coloring. *INFORMS Journal on Computing* 8(4):344–354.

Nemhauser G, Wolsey L (1988) *Integer and Combinatorial Optimization* (Wiley).

Oliveira D, Pessoa A (2020) An improved branch-cut-and-price algorithm for parallel machine scheduling problems. *INFORMS Journal on Computing* 32(1):90–100.

Pecin D, Pessoa A, Poggi M, Uchoa E (2017) Improved branch-cut-and-price for capacitated vehicle routing. *Mathematical Programming Computation* 9(1):61–100.

Pessoa A, Sadykov R, Uchoa E, Vanderbeck F (2018) Automation and combination of linear-programming based stabilization techniques in column generation. *INFORMS Journal on Computing* 30(2):339–360.

Pessoa A, Sadykov R, Uchoa E, Vanderbeck F (2020) A generic exact solver for vehicle routing and related problems. *Mathematical Programming* 183(1):483–523.

Pessoa A, Uchoa E, De Aragão MP, Rodrigues R (2010) Exact algorithm over an arc-time-indexed formulation for parallel machine scheduling problems. *Mathematical Programming Computation* 2(3):259–290.

Righini G, Salani M (2008) New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks: An International Journal* 51(3):155–170.

Ropke S, Cordeau JF, Laporte G (2007) Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks: An International Journal* 49(4):258–272.

Sadykov R, Uchoa E, Pessoa A (2021) A bucket graph–based labeling algorithm with application to vehicle routing. *Transportation Science* 55(1):4–28.

Solomon MM (1987) Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research* 35(2):254–265.

Tang Z (2021) *Theoretical and Computational Methods for Network Design and Routing*. Ph.D. thesis, Carnegie Mellon University.

Tang Z, van Hoeve WJ (2024) Dual Bounds from Decision Diagram-Based Route Relaxations: An Application to Truck-Drone Routing. *Transportation Science* 58(1):257–278.

Toth P, Vigo D (2014) *Vehicle Routing: Problems, Methods, and Applications* (SIAM), second edition.

Uchoa E, Pecin D, Pessoa A, Poggi M, Vidal T, Subramanian A (2017) New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research* 257(3):845–858.

van Den Akker JM, Hoogeveen JA, van de Velde SL (1999) Parallel machine scheduling by column generation. *Operations research* 47(6):862–872.

van Hoeve W (2020) Graph coloring lower bounds from decision diagrams. Bienstock D, Zambelli G, eds., *Integer Programming and Combinatorial Optimization - 21st International Conference, IPCO 2020, London, UK, June 8-10, 2020, Proceedings*, volume 12125 of *Lecture Notes in Computer Science*, 405–418 (Springer).

van Hoeve WJ (2022) Graph coloring with decision diagrams. *Mathematical Programming* 192(1):631–674.

van Hoeve WJ (2024) An Introduction to Decision Diagrams for Optimization. *INFORMS TutORials in Operations Research* (INFORMS).

van Hoeve WJ, Tang Z (2022) Column "Elimination": Dual Bounds From Decision Diagram-based Route Relaxations. *INFORMS Computing Society Conference*.

Vanderbeck F (2005) Implementing Mixed Integer Column Generation. Desaulniers G, Desrosiers J, Solomon M, eds., *Column Generation*, 331–358 (Springer).

Wang C, Wang Y, Wang Y, Wu CT, Yu G (2019) muSSP: Efficient Min-cost Flow Algorithm for Multi-object Tracking. *Advances in Neural Information Processing Systems*, 425–434.

Wolsey LA (1977) Valid inequalities, covering problems and discrete dynamic programs. *Annals of Discrete Mathematics*, volume 1, 527–538 (Elsevier).

Table 1 The performance of column elimination on VRPTW instances from Gehring and Homberger (2002) with 400 and 600 locations. We bold the instances where column elimination outperforms VRPSolver.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
C2_4_1	4100.3	4100.3	1	852	4087.63	17	310	1384	0	3600
C2_4_10	3665.1	3647.88	1	3600	3390.76	1	185	2479	0	3600
C2_4_2	3914.1	3900.22	1	3600	3804.51	1	170	2574	0	3600
C2_4_3	3755.2	3723.96	1	3600	3325.44	1	96	1468	0	3600
C2_4_4	3523.7	3486.12	1	3600	2718.37	1	70	748	0	3600
C2_4_5	3923.2	3923.2	1	971	3827.97	1	408	5653	0	3600
C2_4_6	3860.1	3860.1	1	2466	3699.51	1	309	4455	0	3600
C2_4_7	3870.9	3870.9	1	1483	3699.63	1	313	4385	0	3600
C2_4_8	3773.7	3770.24	1	3600	3557.85	1	265	3680	0	3600
C2_4_9	3842.1	3806.45	1	3600	3581.76	1	276	3671	0	3600
C2_6_1	7752.2	7719.46	1	3600	7691.56	3	541	1995	0	3600
C2_6_10	7123.9	6340.81	1	3600	**6526.6**	1	162	3253	0	3600
C2_6_2	7471.5	7075.15	1	3600	**7210.15**	1	130	2244	0	3600
C2_6_3	7215	4670.06	1	3600	**6181.3**	1	63	1072	0	3600
C2_6_5	7553.8	7540.44	1	3600	7269.61	1	304	6003	0	3600
C2_6_6	7449.8	7400.61	1	3600	7020.32	1	269	5470	0	3600
C2_6_7	7491.3	6294.69	1	3600	**7014.31**	1	258	5282	0	3600
C2_6_8	7303.7	7223.09	1	3600	6811.1	1	248	5038	0	3600
C2_6_9	7303.2	5754.15	1	3600	**6841.19**	1	195	3972	0	3600

Table 2 The performance of column elimination on three difficult VRPTW instances.

Instance		VRPSolver			Column Elimination				
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Time (s)
C1_10_5	42434.8	42434.8	1	1227	42434.8	3	810	80558	9986
C1_8_5	25138.6	25138.6	1	737	25138.6	3	220	2034	1848
C2_10_1	16841.1	16837.4	1	12000	16841.1	24	120	3432	8570

Table 3 Summary statistics to compare the performance of column elimination and VRPSolver. 'Gap' is the average optimality gap over the set of instances and 'Missing' is the number of instances for which the solver failed to produce a dual bound.

	VRPSolver		Column Elimination	
Instance Class	Gap (%)	Missing	Gap (%)	Missing
C1	0.3	4	1.8	0
C2	3.4	21	8.0	4
R1	0.6	11	8.0	23
R2	0.6	34	54.7	31
RC1	1.0	6	9.9	13
RC2	3.6	34	61.7	36

Table 4 The performance of column elimination on the five graph multicoloring instances that it closes.

Instance				GM			Column Elimination				
Name	n	m	ω	LB	UB	Time (s)	LB	UB	CEIt	CR	Time (s)
COG-gesa2-o	192	144	12	12	13	3600	12	**12**	2	0	0
COG-misc07	410	2928	36	36	39	3600	36	**36**	103	378	151
COG-nsrand-ipx	13240	69510	30	-	-	3600	30	**30**	2	0	5
COG-opt1217	1536	6528	26	-	-	3600	26	**26**	2	0	14
COG-rout	560	2940	30	30	32	3600	30	**30**	2	0	0

Table 5 The performance of column elimination on 14 PDPTW instances from Li and Lim (2001) that it solves that have not been reported on by an exact solver.

Instance		Column Elimination				
Name	UB	LB	CEIt	CESIt	CR	Time (s)
LC1_4_1	7152.06	**7152.06**	8	30	152	15
LC1_4_5	7150.0	**7150.0**	8	50	435	54
LC1_4_6	7154.02	**7154.02**	16	80	1218	1349
LC1_4_7	7149.43	**7149.43**	11	90	1583	712
LC1_6_1	14095.64	**14095.6**	9	40	328	71
LC1_6_5	14086.3	**14086.3**	8	90	1307	358
LC1_6_6	14090.79	**14090.8**	8	150	2969	735
LC1_6_7	14083.76	**14083.8**	9	120	2791	2310
LC1_8_1	25184.38	**25184.4**	8	50	591	92
LC1_8_5	25211.22	**25211.2**	8	100	2398	828
LC1_8_6	25164.25	**25164.3**	12	210	4882	3009
LC2_2_1	1931.44	**1931.44**	16	100	478	86
LC2_4_1	4116.33	**4116.33**	10	150	1181	563
LR1_4_1	10639.75	**10639.8**	9	530	3906	437

Appendix A: Dynamic Program Relaxation Example

We give an example of a dynamic program relaxation that is not a state-space relaxation. First, we give the formal definition of a state-space relaxation. We will use a formal definition similar to the one in Christofides et al. (1981). Given (S_1, h_1, c_1, G_1), define a state-space relaxation to be (S_2, h_2, c_2, G_2) that meets the following requirements. First, $|S_2| < |S_1|$. Second, there exists a mapping function $\mu : S_1 \to S_2$ such that for each $s_2 \in S_1$, for all $(s_1, u) \in h_1^{-1}(\{s_2\})$, $(\mu(s_1), u) \in h_2^{-1}(\{\mu(s_2)\})$, where we define $h^{-1}(\{s_2\}) = \{(s_1, d) : h((s_1, d)) = s_2\}$ as the preimage of $s_2 \in S$, not to be confused with an inverse function. Third, for every $s_1 \in S_2$, $c_2(s_1, u) = \min_{\{s_3 \in S_1 | \mu(s_3) = u, \mu(h_1(s_3, u)) = h_2(s_1, u)\}} \{c_1(s_3, u)\}$.

We give an example of when a dynamic program relaxation is not a state-space relaxation in Proposition 2, using data for an example CVRP instance in Figure 6. The costs are not relaxed in either dynamic program, so we only argue about the solution sets complying with the definitions.

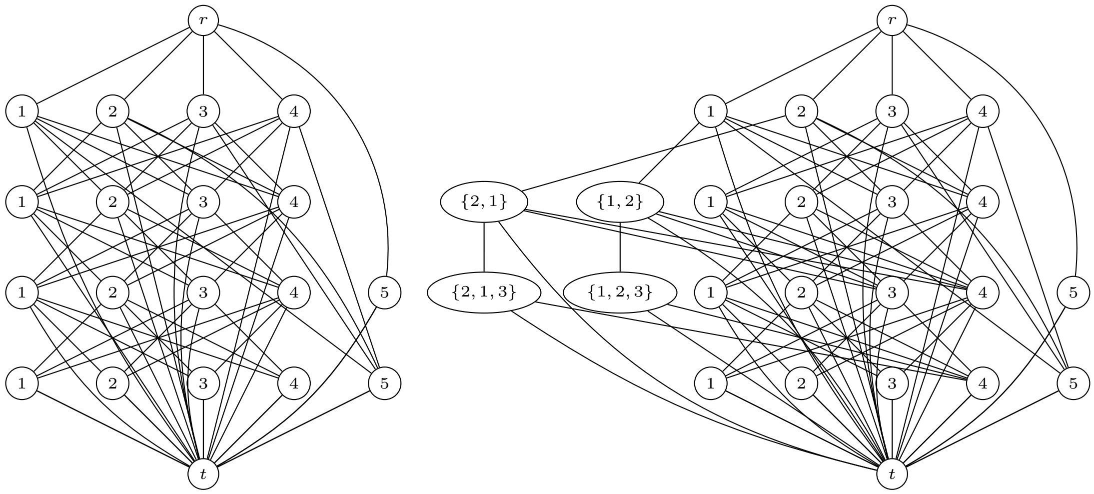
Proposition 2. *The dynamic program relaxation represented by the state-transition graph on the right side of Figure 5 is not a state-space relaxation.*

Proof For sake of contradiction assume the dynamic program relaxation is a state-space relaxation defined by a mapping μ. It must be the case that $\mu(r) = r$, which implies $\mu((\{1\}, 1, 1)) = (\{1\}, 1, 1)$ and $\mu((\{2\}, 1, 2)) = (\{2\}, 1, 2)$. This implies $\mu((\{1, 2\}, 2, 2)) = (\{1, 2\}, 2, 2)$ and $\mu((\{2, 1\}, 2, 1)) = (\{1\}, 2, 1)$. Finally, this implies $\mu((\{1, 2, 3\}, 3, 3)) = (\{1, 2, 3\}, 3, 3)$, which would require $h((\{2, 1\}, 2, 1), 3) = (\{1, 2, 3\}, 3, 3)$, but in the dynamic program relaxation shown $h((\{2, 1\}, 2, 1), 3) = (\{2, 1, 3\}, 3, 3)$ (the states labelled $(\{1, 2, 3\}, 3, 3)$ and $(\{2, 1, 3\}, 3, 3)$ are distinct even though they represent the same information), which is a contradiction. *Q.E.D.*

Appendix B: Application Models

B.1. Vehicle Routing Problem with Time Windows (VRPTW)

The problem $\mathcal{P}$ has the same form as the CVRP, but the set $\mathcal{S}$ is further restricted to routes that respect the time window constraints. To create P, we extend the model for the CVRP as follows. We augment the states of the dynamic program to consider time. The states become tuples (NG, w, v, τ) where τ is the current time. The initial state is $r = (\varnothing, 0, 0, 0)$. Given a state $s = (\text{NG}, w, v, \tau)$ and element $u \in U$ such that $u \neq 0$, $u \notin \text{NG}$, $w + q_u \leq Q$, and $\tau + \ell_{v,u} \leq l_u$, the transition function becomes $h(s, u) = (\text{NG} \cup \{u\}, w + q_u, u, \max(\tau + \ell_{v,u}, e_u))$. Otherwise if $u = 0$ and $\tau + \ell_{v,0} \leq l_0$, define $h(s, 0) = t$. Otherwise $h(s, u) = -1$. The cost function is the same



State-space relaxation. Dynamic program relaxation.

Figure 5 The state-transition graph on the left is a state-space relaxation of a dynamic program, and the state-transition graph on the right is a dynamic program relaxation created by removing some feasible sequences from the state-space relaxation on the left.

Locations $V = \{0, 1, 2, 3, 4, 5\}$

Depot $= 0$

Demands $q_1 = q_2 = q_3 = q_4 = 1$, $q_5 = 3$

Number of vehicles $K = 2$

Vehicle capacity $Q = 4$

l_{ij}	0	1	2	3	4	5
0	0	12	10	11	10	11
1	12	0	2	1	22	23
2	10	2	0	1	20	21
3	11	1	1	0	21	22
4	10	22	20	21	0	1
5	11	23	21	22	1	0

Figure 6 Input data for a CVRP instance.

as for the CVRP. The constraints G are the same as for the CVRP, but without the constraint requiring K vehicles.

We relax the model by creating a dynamic programming relaxation w.r.t. P. We use an ng-route relaxation and also consider relaxing the time windows and/or vehicle capacity. To relax the time windows, we use a 'bucketing' idea from the column generation literature (Sadykov et al. 2021). When a transition would create a state with time value τ, round down τ to the nearest multiple of $\Delta \in \mathbb{Z}$. To relax the load values, for certain instances we set all states to have load 0. This way, we only create states that remember load when conflicts are refined. We use a binary value κ to indicate if capacity should be relaxed or not. We add a counter c

to all states to maintain an acyclic state-transition graph (Horn 2021). We use an upper bound denoted U on the number of locations in a route, because the relaxation can create many long infeasible routes. We calculate U by using two greedy methods based on summing the smallest loads up to Q and summing the shortest distances with service times compared to l_0.

Formally, we construct an initial dynamic programming relaxation $P' = (S', h', c', G')$ w.r.t. P. Each state is a tuple $(\mathrm{NG}, w, v, \tau, c)$ with the initial state being $r_1 = (\emptyset, 0, 0, 0, 0)$. The set of states S' is implicitly defined by defining the following transition function h'. Given a state $s = (\mathrm{NG}, w, v, \tau, c)$ and label $u \in U$ such that $u \neq 0$, $u \notin \mathrm{NG}$, $w + q_u \leq Q$, $\tau + \ell_{v,u} \leq l_u$, and $c < U$, let the transition function be $h'(s,u) = ((\mathrm{NG} \cup \{u\}) \cap N_u, (w + q_u) * (1 - \kappa), u, \max(\lfloor \frac{\tau + \ell_{v,u}}{\Delta} \rfloor * \Delta, e_v), c + 1)$. Otherwise, when $u = 0$ and $\tau + \ell_{v,0} \leq l_0$, define $h'(s, 0) = t$. Otherwise $h'(s,u) = -1$. Then, we keep the same cost function $c' = c$ and the same additional cost function $G' = G$.

B.2. Graph Multicoloring

The problem $\mathcal{P}$ is formed by an element set $U = \{1, ..., |V|\}$, a set $\mathcal{S}$ containing all independent sets in $\mathcal{G}$ as sequences with strictly increasing values, a constant cost function $f = 1$, and for a subset of sequences X, $C(X) = 1$ if and only if all vertices are contained in at least one sequence. We represent the constraint function as a tuple $(\gamma_j, =, b_j)$ for each $j = 1, ..., |V|$ such that $\gamma_j(x) = [\![j \in x]\!]$.

We model the problem with a dynamic program $P = (S, h, c, G)$. The dynamic program will depend on an ordering of the vertices $\{1, \ldots, |V|\}$, similar to the one in van Hoeve (2022). In our experiments, we use a variable ordering called 'min width' from Karahalios and van Hoeve (2022). Let each state $s = (\mathrm{NG})$ contain a 'no-good' subset of vertices NG that can no longer be included in an independent set. The initial state is $r = (\varnothing)$. The transition function given a state $s = (\mathrm{NG})$ and decision i such that $i \notin \mathrm{NG}$ is $h(s,i) = (\mathrm{NG} \cup N_i \cup \{1, \ldots, i\})$, where N_i is the set of neighbors of vertex i, and all vertices with smaller index are included in the updated NG to break symmetries. The dynamic program differs from the one in van Hoeve (2022), because in that work the set of decisions was binary and each transition corresponded to selecting a vertex to be in the independent set or not. The cost function is $c(r, i) = 1$ for all $i \in U$, and $c(s, i) = 0$ when $s \neq r$ for all $i \in U$. Second, we construct G. For each $j = 1, ..., m$, we create an additional cost function $g_j((s,u)) = [\![j = u]\!]$.

We relax the model by creating a dynamic programming relaxation $P' = (S', h', c', G')$ w.r.t. P. Each state maintains a 'no-good' subset of vertices $s = (\mathrm{NG})$ and the initial state is $r = (\varnothing)$. The dynamic program only 'remembers' the latest decision. Formally, given a state $s = (\text{NG})$ and feasible transition $u \in U$ such that $u \notin \text{NG}$, define $h'(s,u) = (\{1,\dots,u\} \cup N_u)$. Otherwise, $h'(s,u) = -1$. We keep the same cost function $c' = c$ and additional cost functions $G' = G$.

In our experiments, we use a common preprocessing rule to simplify the graph $\mathcal{G}$ before setting up the model. We remove a vertex if the sum of the weights of its neighbors plus its own weight is smaller than a lower bound on the optimal solution value. For multicoloring, we use an initial maximum (weighted) clique from Gualandi and Malucelli (2012) as an initial lower bound for this preprocessing.

B.3. Pickup-and-Delivery Problem with Time Windows (PDPTW) and Sequential Ordering Problem (SOP)

The problem $\mathcal{P}$ has the same form as the VRPTW, but the sequences in $\mathcal{S}$ must also follow the precedence constraints. We define the set of precedence locations for each location $u \in V$ as Π_u. So, for $u \in D$, $\Pi_u = \{u - n\}$, and $\Pi_u = \emptyset$ otherwise. We define the transition function as follows. Each state is a tuple $s = (\text{NG}, w, v, \tau)$ and the initial state is $r = (\text{NG}, w, v, \tau)$. Given a state $s = (\text{NG}, w, v, \tau)$ and element $u \in U$ such that $u \neq 0$, $u \notin \text{NG}$, $\Pi_u \subseteq \text{NG}$, $w + q_u \leq Q$, and $\tau + \ell_{v,u} \leq l_u$, let $h(s,u) = (\text{NG} \cup \{u\}, (w + q_u) * (1 - \kappa), u, \lfloor \frac{\max(\tau + \ell_{v,u}, e_u)}{\Delta} \rfloor * \Delta, c+1)$. Otherwise, when $u = 0$ and $\tau + \ell_{v,0} \leq l_0$, define $h(s,0) = t$. Otherwise $h(s,u) = -1$. The cost function and the additional costs are the same as for the VRPTW.

We relax the model in a similar way to the VRPTW, while also relaxing precedence constraints. To relax the model, we create a dynamic programming relaxation $P' = (S', h', c', G')$ w.r.t. P. Again, we add a counter c to each state, so each state has the form $s = (\text{NG}, w, v, \tau, c)$. The initial state is $r = (\emptyset, 0, 0, 0, 0)$. Then, for a state $s = (\text{NG}, w, v, \tau, c)$ and transition u such that $u \neq 0$, $u \notin \text{NG}$, $w + q_u \leq Q$, $\tau + \ell_{v,u} \leq l_u$, and $c < |V|$, let $h'(s,u) = ((\text{NG} \cup \{u\}) \cap N_u, (w + q_u) * (1 - \kappa), u, \max(\lfloor \frac{\tau + \ell_{v,u}}{\Delta} \rfloor * \Delta, e_u), c+1)$. When $u = 0$, let $h'(s,u) = t$. Otherwise $h'(s,u) = -1$. We use the same cost function $c_1 = c$ and additional cost functions $G' = G$.

Similar to other works in the vehicle routing literature, we aim to solve instances based on distances that are rounded to the thousandths place. To do this, we start by rounding distances to the hundredths place and solving the instance. Then, we keep the P_i at termination, update the distances to be rounded to the thousandths place, and solve the instance again. We use best-known solution values from the literature, which sometimes use different rounding, which accounts for small discrepancies in the full table of results in Appendix N. We model the SOP in this way, with the additional constraint that a solution has one sequence.

Appendix C: Capacitated Vehicle Routing Problem (CVRP) Formulation

The model F to solve the CVRP is formulated as follows:

$$F: \quad \min \sum_{a\in\mathcal{A}} c_a y_a \tag{9}$$

$$\text{s.t.} \sum_{a\in\delta^+(r)} y_a = K \tag{10}$$

$$\sum_{(s,u)\in\mathcal{A}:u=j} y_a = 1 \qquad \forall j = 1,...,m \tag{11}$$

$$\sum_{a\in\delta^+(u)} y_a - \sum_{a\in\delta^-(u)} y_a = 0 \qquad \forall u \in \mathcal{N}\setminus\{r,t\} \tag{12}$$

$$y_a \in \{0,1\} \qquad \forall a \in \mathcal{A}. \tag{13}$$

The objective function (9) minimizes the sum of all arc costs. The 'flow conservation' constraints (12) ensure that the solution is a collection of labeled r-t paths, or single feasible routes. Constraint (10) enforces that exactly K units of flow originate from r and thus K routes are used. Constraints (11) ensure that all locations are visited once. The binary constraints (13) complete the formulation.

Appendix D: Variable Fixing

Variable fixing is an important method to improve the performance of column elimination. Variable fixing is an acceleration method used in integer programming (Nemhauser and Wolsey 1988) and constraint programming (Focacci et al. 1999) to prove that a variable must equal its lower bound or upper bound in an optimal solution. For integer programming, the proof requires a primal bound and a feasible dual solution to the linear programming relaxation. For column elimination, we use a variable fixing algorithm that considers one arc $a\in\mathcal{A}$ at a time and reasons about all $r-t$ paths that traverse the arc. Theorem 6 generalizes the arc fixing theorem from Karahalios and van Hoeve (2023), which is based on Irnich et al. (2010).

Let ν be a feasible solution to the dual of $LP(F)$ such that each ν corresponds to $\mathcal{G}$. For each arc $a\in\mathcal{A}$ we define a 'reduced cost distance' $rc(y_a) = c_a - \sum_{j=1}^m g_j(a)\nu_j$. For each node $u\in\mathcal{N}$, we define $sp_u^{\downarrow}$ as the shortest r-u path in the state-transition graph of P with respect to the reduced cost distances, and similarly define $sp_u^{\uparrow}$ to be the shortest u-t path in the state-transition graph of P.

THEOREM 6. *Consider arc $a = (v_1, v_2)\in\mathcal{A}$. Let $v(\nu)$ be the solution value of ν to the dual of $LP(F)$, and let UB an upper bound on the optimal solution value for F. If $v(\nu) + sp_{v_1}^{\downarrow} + sp_{v_2}^{\uparrow} + rc(y_a) > UB$, then arc a can be fixed to have flow 0 in F.*

Proof Consider the following integer program that is equivalent to F, created by enumerating the solutions to $\mathcal{S}$, where $\mathcal{A}_x$ is the set of arcs in the solution x.

$$(IP) \quad \min \sum_{x \in \mathcal{X}} z_x f(x) \tag{14}$$

$$\text{s.t.} \sum_{x \in \mathcal{X}} z_x \sum_{a \in \mathcal{A}_x} g_j(a) \circ_j b_j \qquad \forall\, j \in \{1, \ldots, m\} \tag{15}$$

$$z \in \mathbb{Z}_+^{|\mathcal{X}|} \tag{16}$$

Given ν and a solution x, in the linear program relaxation of IP, the variable z_x has reduced cost $rc(x) = f(x) - \sum_{j=1}^m \nu_j \sum_{a \in \mathcal{A}_x} g_j(a)$. Each x corresponds to a path $p = \{a_1, \ldots, a_l\}$ in $\mathcal{D}$, so $rc(x)$ can be decomposed into $rc(x) = \sum_{a \in \mathcal{A}_x} rc(y_a)$. For all p that contain arc a, let p' be the path that corresponds to the route x with lowest reduced cost. Denote the lowest reduced cost as $rc'(x') = sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(y_a)$. Now for sake of contradiction assume an optimal solution to F has $y_a = 1$. Then some solution x'' in $\mathcal{D}$ that contains arc a will be in the solution X to F, which is equivalent to $z_{x''} = 1$ in IP. To construct the remainder of an optimal solution to the linear programming relaxation of IP, we can solve the linear programming relaxation with the constraints defined by G adjusted to have the values contributed from x'' removed. Then, ν remains feasible to the dual of this updated problem and has value $v(\nu) - \sum_{j=1}^m \nu_j \sum_{a \in \mathcal{A}_{x''}} g_j(a)$. So, combining this feasible dual with the cost of x'' gives a valid lower bound on IP as $v(\nu) - \sum_{j=1}^m \nu_j \sum_{a \in \mathcal{A}_{x''}} g_j(a) + f(x'')$. This contradicts UB. Specifically, $v(\nu) - \sum_{j=1}^m \nu_j \sum_{a \in \mathcal{A}_{x''}} g_j(a) + f(x'') = v(\nu) + rc(x'') \geq v(\nu) + rc(x') \geq v(\nu) + sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(y_a) > UB$. Q.E.D.

Appendix E: Convergence of Column Elimination with Subgradient Descent

It is not straightforward to analyze the convergence of column elimination with subgradient descent. Given F_i, subgradient descent will converge to an optimal dual solution for $LP(F_i)$ if a divergent series step-length is used (Anstreicher and Wolsey 2009). However, an optimal primal solution is needed to determine if any conflicts need to be refined. Subgradient descent produces a sequence of solutions to the Lagrangian relaxation whose average converges to an optimal primal solution, but this does not theoretically guarantee that an optimal primal solution is found (Anstreicher and Wolsey 2009).

So, to evaluate the convergence of the algorithm in practice, we consider the following example. We apply column elimination and column elimination with subgradient descent to solve the instance C1_2_5 of the

Vehicle Routing Problem with Time Windows (VRPTW) from Gehring and Homberger (2002), which we define in the next section. For each algorithm, we plot the sequence of dual solutions obtained at each iteration, converted into three dimensions using a principal component analysis method. Figure 7(a) shows that the optimal dual solutions of column elimination can greatly change from one iteration to the next, possibly indicating degeneracy, which can hinder column generation. In contrast, Figure 7(b) shows a smooth path of dual solution values, indicating that each step of subgradient descent is in the general direction of the optimal dual solution to $LP(F)$ that the algorithm finds upon termination.

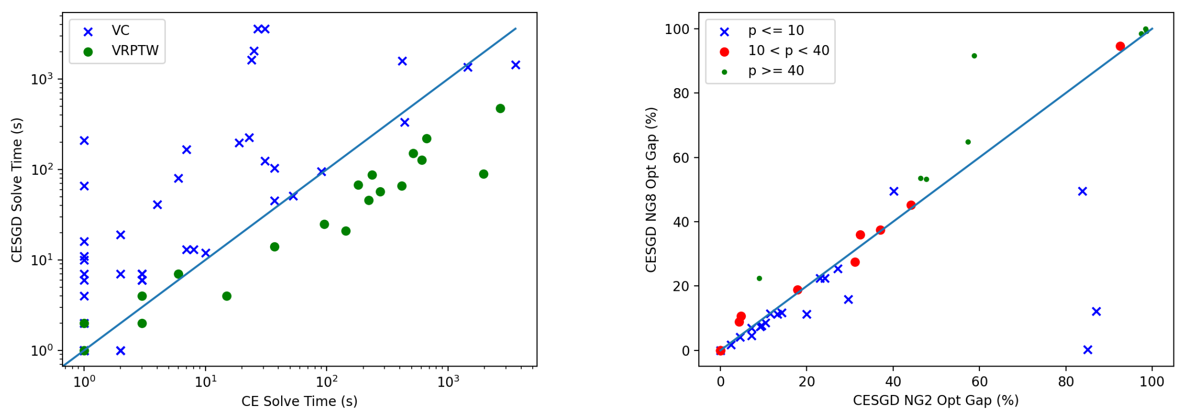
a. CPLEX b. Lagrangian Method

Figure 7 **Sequences of dual solutions obtained by running a column elimination and column elimination with subgradient descent to solve the VRPTW instance C1_2_5. The dual solutions are plotted in three dimensions using the Python package 'sklearn'.**

Appendix F: Evaluating Column Elimination with Subgradient Descent

We give insights into when column elimination with subgradient descent will outperform column elimination. To do this, we apply each algorithm to solve the vertex coloring instances from Johnson and Trick (1996) and the VRPTW instances from Gehring and Homberger (2002) and Solomon (1987). We choose these two applications, because they differ in the following way. The primal and dual solutions to F_i and F_{i+1} for some iteration i can be greatly different for vertex coloring. However, for the VRPTW, the primal and dual solutions can remain very similar. This property affects the performance of column elimination with subgradient descent for two reasons. Firstly, it changes the likelihood that refinements of conflicts at the current iteration will benefit future iterations. Secondly, the steps of subgradient descent are more likely to be in the direction of the optimal solutions to F. So, we hypothesize that column elimination with subgradient descent will perform well for the VRPTW, but not for vertex coloring problem.

We show a plot that validates our hypothesis. For instances that both column elimination and column elimination with subgradient descent solver, we plot the time it takes for column elimination and column elimination with subgradient descent to solve each instance in Figure 8(a). For vertex coloring, column elimination solves instances on average 195 seconds faster than column elimination with subgradient descent. For the VRPTW, the column elimination with subgradient descent solves instances on average 330 seconds faster than column elimination. For vertex coloring, column elimination solves 13 additional instances and column elimination with subgradient descent solves no additional instances. For the VRPTW, column elimination with subgradient descent solves two additional instances and column elimination solves one additional instance.



a. Subgradient Descent b. Initial Relaxations for SOP

Figure 8 **a) The time taken to solve VRPTW and vertex coloring (VC) instances using column elimination (CE) and column elimination with subgradient descent (CESGD). The axes use log scales. b) The optimality gap achieved when using column elimination with subgradient descent starting with an initial ng-route relaxation with $\rho = 2$ (CESGD NG2) and an initial ng-route relaxation with $\rho = 8$ (CESGD NG8) for the SOP with p% precedence constraints.**

Appendix G: Sensitivity to Initial Relaxations

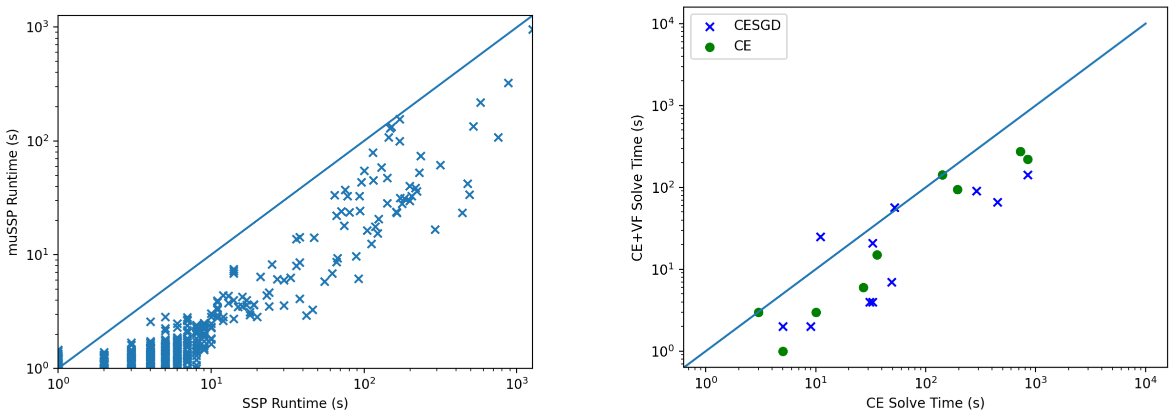
We evaluate the sensitivity of column elimination to the initial relaxation. To do this, we use column elimination with subgradient descent and two different initial relaxations to solve the TSPLIB SOP instances (Ascheuer et al. 2000). The two initial relaxations are ng-route with $\rho = 2$ and $\rho = 8$. We aim to test the behavior of the following general tradeoff. A stronger initial relaxation can reduce the number of conflict refinements needed for column elimination to solve the problem, but it can increase the time needed to solve $LP(F_i)$ at each iteration. In the context of the SOP, we consider that an initial relaxation relaxes precedence constraints and the constraint that the solution is an elementary path. We hypothesize that a stronger ng-route relaxation will capture the constraint that a solution is an elementary path, but not capture the precedence constraints because the NG sets are different for each location, so larger NG sets do not necessarily encode precedence constraints for solutions that have two locations appearing far apart in the ordering.

We show a plot that gives insight into this hypothesis. We plot the optimality gaps achieved at termination when starting with the $\rho = 2$ and $\rho = 8$ initial relaxations in Figure 8(b). The instances are partitioned based on the percent of values in the distance matrix that are indicate a precedence, which we denote at p. The groups are $p \leq 10$, $10 < p < 40$, and $p \geq 40$. The instances in the group $p \leq 10$ tend to have a smaller number of vertices than the instances in the group with $p \geq 40$. The average number of locations is 63 and 263 respectively. The plot shows that for instances with a low percent of precedences, starting with the $\rho = 8$ initial relaxation is beneficial. For these instances, the size of the initial relaxation for $\rho = 8$ is small enough that column elimination can run for many iterations; when using $\rho = 8$ the median number of iterations solved is 3507 and when using $\rho = 2$ the median is 19389 iterations. In comparison, the plot shows that for instances with a high percent of precedences, starting with the $\rho = 2$ initial relaxation is beneficial. For these instances, when using $\rho = 8$, the median number of iterations solved is only 206, and when using $\rho = 2$ the median is 1038 iterations.

We give a full table of results with a comparison to a state-of-the-art solver in Appendix O. For high level statistics, column elimination solved x instances to optimality, failed to compute a lower bound for 6 instances, and returned an average optimality gap of 36.3% for the remaining instances. In comparison, the state-of-the-art solver (Libralesso et al. 2020) solved 8 instances to optimality and returned an average optimality gap of 87% for the remaining instances.

Appendix H: Impact of Using Variable Fixing

We evaluate the effect of using variable fixing during column elimination. We use column elimination with and without subgradient descent, and with and without variable fixing, to solve the VRPTW instances from Solomon (1987). We plot the run times of instances solved by both methods in Figure 9(a). Variable fixing allows column elimination with subgradient descent to solve one additional instance and solved instances on



a. Impact of muSSP b. Impact of Variable Fixing

Figure 9 **a) The difference in solve time of $L(\lambda)$ between SSP and muSSP for solving VRPTW instances with column elimination. For both plots, the axes use log scales. b) The runtime to solve CVRP instances using column elimination and column elimination with subgradient descent both with and without variable fixing.**

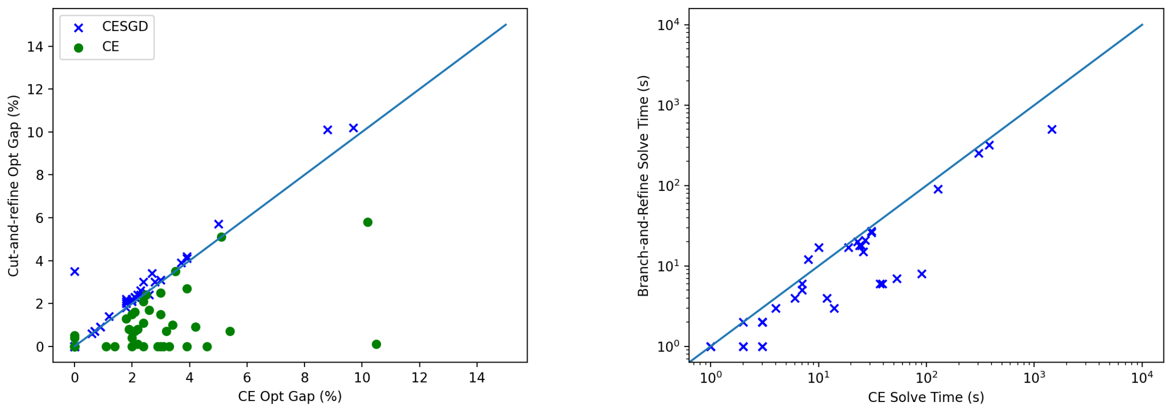
average 106 seconds faster than without variable fixing. Variable fixing allows column elimination without subgradient descent to solve 4 additional instances and solved instances on average 110 seconds faster than without variable fixing. These are the first experimental results for using variable fixing during column elimination with subgradient descent by checking dual feasibility and repairing infeasible duals.

Appendix I: Impact of Using Minimum Update SSP

We evaluate the impact of the minimum update successive shortest paths (muSSP) instead of SSP during column elimination with subgradient descent. We use column elimination with subgradient descent to solve ten CVRP instances from Uchoa et al. (2017) each with a different number of locations. At each iteration, we solve $L(\lambda)$ using both muSSP and SSP. We plot the runtimes in Figure 9(b). For smaller instances, there is not much impact, but for larger instances muSSP can greatly improve performance. Using muSSP solves the subproblem on average 3.7 times faster than using SSP.

Appendix J: Evaluating Cut-and-refine

We evaluate the performance of cut-and-refine compared to column elimination. We implement cut-and-refine for the CVRP using the framework in Figure 3. We apply column elimination without adding any cutting planes and cut-and-refine on the CVRP instances from `http://vrp.atd-lab.inf.puc-rio.br/index.php/`



a. Cut-and-refine b. Branch-and-refine

Figure 10 **a) The runtime to solve VRPTW instances using column elimination with and without subgradient descent, with and without cuts. b) The runtime to solve vertex coloring instances using column elimination (CE) and branch-and-refine. For this plot the axes use log scales.**

`en/` in classes A,B,M,E,F, and P. We also apply column elimination with subgradient descent without any cutting planes and cut-and-refine with subgradient descent to the same instances. We choose to remove the constraint that a solution must use K vehicles, as this is the case for the large-scale VRPTW instances that we will evaluate when comparing to the state-of-the-art. Similar experiments that include the constraint on the number of vehicles are shown in Karahalios and van Hoeve (2023).

We plot the optimality gaps achieved at termination for both column elimination without cuts, column elimination with subgradient descent without cuts, cut-and-refine, and cut-and-refine with subgradient descent in Figure 10(a). The plot shows that cut-and-refine achieves better optimality gaps than column elimination without cutting planes. However, the plot also shows that cut-and-refine with subgradient descent does not improve the performance of column elimination with subgradient descent without cutting planes.

The cuts that we implement are categorized as ‘robust’ in the column generation literature. This name comes from the fact that they do not ‘complicate’ the pricing problem. Much work has been done in the column generation literature to incorporate ‘non-robust’ cuts, which do complicate the pricing problem (Pecin et al. 2017). For many instances of vehicle routing problems, the addition of non-robust cuts called subset-row inequalities is important to closing the instances, which currently gives these methods an advantage over column elimination. Non-robust cuts have not been effectively added in column elimination, although ideas similar to the one in Hoogendoorn and Dalmeijer (2025) may work.

Appendix K: Evaluating Branch-and-refine

We evaluate the performance of branch-and-refine. We implement branch-and-refine for the vertex coloring problem. We use Zykov branching, which chooses two nonadjacent vertices and defines one branch by contracting these vertices and the other branch by adding an edge between the vertices (Corneil and Graham 1973). We choose the two nonadjacent vertices with the highest sum of their degrees, using an ordering of the vertices as a tie breaker. We terminate the algorithm when solving a subproblem if there has not been an improvement to the lower bound for 30 seconds. We choose the subproblem with the greatest lower bound as the one to solve next, using the most recently created node as a tie breaker. We solve the DIMACS set of vertex coloring instances using branch-and-refine and column elimination without branch-and-bound (Johnson and Trick 1996).

We plot the runtimes of column elimination and branch-and-refine on instances that both methods solve in Figure 10(b). Branch-and-refine solves instances on average 2.3 times faster than without using branching. Branch-and-refine solves an additional 7 instances, mostly FullIns instances related to Mycielski graphs. Using column elimination without branching solves an additional 8 instances, mostly larger instances that require more conflict refinements.

We also compare to a state-of-the-art solver, given the same timeouts and initial upper bounds, and run the instances on the same server Held et al. (2011). Column elimination solves 67 instances to optimality, fails to give a bound on 9 instances, and has an average gap of 33.7% on the remaining instances. The state-of-the-art solver solves 89 instances to optimality, fails to give a bound on 18 instances, and has an average gap of 15.6% on the remaining instances. A full table of results is given in Appendix P.

Appendix L: Vehicle Routing Problem with Time Windows Results

In Table 6 we compare column elimination with VRPSolver on the vehicle routing problem with time windows (VRPTW). For each instance, we list the current best-known upper bound (UB) that each algorithm takes as input. For VRPSolver we report the lower bound (LB), the number of explored branch-and-cut-and-price nodes (Nodes), and the solution time (Time (s)). For column elimination, we report the lower bound (LB), the number of column elimination iterations (CEIt) and column elimination with subgradient descent iterations (CESIt), the number of cuts (Cuts), the number of refinements (CR), and the solution time (Time (s)).

Table 6 A comparison of the performance of VRPSolver from Pessoa et al. (2020) and column elimination for solving VRPTW instances by Gehring and Homberger (2002).

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
C1_10_1	42444.8	42444.8	1	1219	42444.8	4	220	576	0	756
C1_10_10	39816.8	-	-	3600	38158.9	1	89	13090	0	3600
C1_10_2	41337.8	41068.76	1	3600	40078.2	1	154	9126	0	3600
C1_10_3	40064.4	-	-	3600	38245.5	1	25	8468	0	3600
C1_10_4	39434.1	-	-	3600	37825.6	1	19	4699	0	3600
C1_10_5	42434.8	42434.8	1	1227	42355.9	1	423	5405	0	3600
C1_10_6	42437	42437.0	1	1670	41987.5	1	323	12458	0	3600
C1_10_7	42420.4	42305.87	1	3600	41673.7	1	472	13079	0	3600
C1_10_8	41652.1	41062.0	1	3600	39687.8	1	292	19422	0	3600
C1_10_9	40288.4	39508.04	1	3600	38459.2	1	144	15855	0	3600
C1_2_1	2698.6	2698.6	1	11	2698.6	3	10	7	0	1
C1_2_10	2624.7	2624.7	1	218	2519.6	6	1009	12510	0	3600
C1_2_2	2694.3	2694.3	1	29	2681.66	21	2610	7624	0	3600
C1_2_3	2675.8	2675.8	3	338	2610.87	3	871	8836	0	3600
C1_2_4	2625.6	2625.6	1	457	2519.19	1	540	9155	0	3600
C1_2_5	2694.9	2694.9	1	16	2694.9	3	30	89	0	7
C1_2_6	2694.9	2694.9	1	20	2694.9	3	40	225	0	15
C1_2_7	2694.9	2694.9	1	18	2694.9	4	50	220	0	22
C1_2_8	2684	2684.0	1	26	2657.08	21	2617	8763	0	3600
C1_2_9	2639.6	2639.6	1	65	2568.08	10	1382	13212	0	3600
C1_4_1	7138.8	7138.8	1	99	7138.8	3	20	26	0	11
C1_4_10	6825.4	6820.19	1	3600	6631.87	1	365	10586	0	3600
C1_4_2	7113.3	7113.3	7	587	7060.35	2	591	4926	0	3600
C1_4_3	6929.9	6929.9	1	991	6797.26	1	251	7320	0	3600
C1_4_4	6777.7	6769.16	1	3600	6589.6	1	129	9037	0	3600
C1_4_5	7138.8	7138.8	1	134	7138.8	3	50	296	0	63
C1_4_6	7140.1	7140.1	1	204	7140.1	4	90	1066	0	202
C1_4_7	7136.2	7136.2	1	185	7136.2	51	120	1987	0	1959
C1_4_8	7083	7083.0	5	1128	6941.63	1	641	9803	0	3600
C1_4_9	6927.8	6927.8	1	1547	6725.5	1	428	10911	0	3600
C1_6_1	14076.6	14076.6	1	292	14076.6	3	50	75	0	55
C1_6_10	13617.5	13520.25	1	3600	13188.1	1	187	11778	0	3600
C1_6_2	13948.3	13948.3	15	1616	13775.7	1	269	4096	0	3600
C1_6_3	13757	13702.24	1	3600	13368.6	1	110	8562	0	3600
C1_6_4	13538.6	13347.86	1	3600	13090.8	1	64	7567	0	3600
C1_6_5	14066.8	14066.8	1	393	14066.8	3	150	797	0	462
C1_6_6	14070.9	14070.9	1	531	14054.8	11	390	3303	0	3600
C1_6_7	14066.8	14066.8	1	476	14058.1	15	320	2860	0	3600
C1_6_8	13991.2	13967.03	3	3600	13689.5	1	462	10889	0	3600
C1_6_9	13664.5	13649.77	1	3600	13299.7	1	287	12421	0	3600
C1_8_1	25156.9	25156.9	1	761	25156.9	3	80	222	0	187
C1_8_10	24026.7	23640.28	1	3600	23129.2	1	141	12807	0	3600
C1_8_2	24974.1	24910.3	1	3600	24397.0	1	211	7703	0	3600
C1_8_3	24156.1	23865.67	1	3600	23373.3	1	77	9104	0	3600
C1_8_4	23797.3	-	-	3600	22809.0	1	23	6233	0	3600
C1_8_5	25138.6	25138.6	1	737	25138.6	3	210	2012	0	1421

Table 6 Continued

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
C1_8_6	25133.3	25133.3	1	1056	25030.9	1	373	5943	0	3600
C1_8_7	25127.3	25127.3	7	1140	24965.2	1	398	5693	0	3600
C1_8_8	24809.7	24688.04	1	3600	24050.7	1	409	18644	0	3600
C1_8_9	24200.4	23972.45	1	3600	23346.5	1	218	14876	0	3600
C2_10_1	16841.1	-	-	3600	16802.5	2	100	1012	0	3600
C2_10_10	15728.6	-	-	3600	12993.3	1	48	1575	0	3600
C2_10_2	16462.6	-	-	3600	15127.1	1	74	2069	0	3600
C2_10_5	16521.3	-	-	3600	15639.4	1	197	6448	0	3600
C2_10_6	16290.7	-	-	3600	15134.9	1	173	5396	0	3600
C2_10_7	16378.4	-	-	3600	15056.9	1	112	3550	0	3600
C2_10_8	16029.1	-	-	3600	14402.2	1	90	2615	0	3600
C2_10_9	16075.4	-	-	3600	12279.1	1	35	1490	0	3600
C2_2_1	1922.1	1922.1	1	228	1922.1	4	70	24	0	37
C2_2_10	1791.2	1791.2	1	631	1682.31	1	584	4204	0	3600
C2_2_2	1851.4	1851.4	1	387	1821.94	2	604	3440	0	3600
C2_2_3	1763.4	1753.63	3	3600	1661.01	1	259	2062	0	3600
C2_2_4	1695	1666.49	3	3600	1507.88	1	164	1443	0	3600
C2_2_5	1869.6	1869.6	1	276	1837.66	6	1069	7902	0	3600
C2_2_6	1844.8	1844.8	1	249	1784.65	1	799	5883	0	3600
C2_2_7	1842.2	1842.2	1	170	1788.26	1	677	5055	0	3600
C2_2_8	1813.7	1813.7	1	222	1731.34	1	692	4939	0	3600
C2_2_9	1815	1815.0	1	511	1728.5	1	590	4237	0	3600
C2_4_1	4100.3	4100.3	1	852	4087.63	17	310	1384	0	3600
C2_4_10	3665.1	3647.88	1	3600	3390.76	1	185	2479	0	3600
C2_4_2	3914.1	3900.22	1	3600	3804.51	1	170	2574	0	3600
C2_4_3	3755.2	3723.96	1	3600	3325.44	1	96	1468	0	3600
C2_4_4	3523.7	3486.12	1	3600	2718.37	1	70	748	0	3600
C2_4_5	3923.2	3923.2	1	971	3827.97	1	408	5653	0	3600
C2_4_6	3860.1	3860.1	1	2466	3699.51	1	309	4455	0	3600
C2_4_7	3870.9	3870.9	1	1483	3699.63	1	313	4385	0	3600
C2_4_8	3773.7	3770.24	1	3600	3557.85	1	265	3680	0	3600
C2_4_9	3842.1	3806.45	1	3600	3581.76	1	276	3671	0	3600
C2_6_1	7752.2	7719.46	1	3600	7691.56	3	541	1995	0	3600
C2_6_10	7123.9	6340.81	1	3600	**6526.6**	1	162	3253	0	3600
C2_6_2	7471.5	7075.15	1	3600	**7210.15**	1	130	2244	0	3600
C2_6_3	7215	4670.06	1	3600	**6181.3**	1	63	1072	0	3600
C2_6_5	7553.8	7540.44	1	3600	7269.61	1	304	6003	0	3600
C2_6_6	7449.8	7400.61	1	3600	7020.32	1	269	5470	0	3600
C2_6_7	7491.3	6294.69	1	3600	**7014.31**	1	258	5282	0	3600
C2_6_8	7303.7	7223.09	1	3600	6811.1	1	248	5038	0	3600
C2_6_9	7303.2	5754.15	1	3600	**6841.19**	1	195	3972	0	3600
C2_8_1	11631.9	-	-	3600	11616.6	9	150	1023	0	3600
C2_8_10	10946	-	-	3600	9898.34	1	133	2961	0	3600
C2_8_2	11394.5	-	-	3600	10777.8	1	108	3089	0	3600
C2_8_3	11138.1	-	-	3600	2920.84	1	13	1492	0	3600
C2_8_5	11395.6	-	-	3600	10911.9	1	248	6313	0	3600
C2_8_6	11316.3	-	-	3600	10624.8	1	248	6295	0	3600
C2_8_7	11332.9	-	-	3600	10613.4	1	185	4694	0	3600
C2_8_8	11133.9	-	-	3600	10280.1	1	184	4548	0	3600
C2_8_9	11140.4	-	-	3600	10242.4	1	147	3634	0	3600

Table 6 Continued

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
R1_10_1	53046.5	52756.54	1	3600	51447.2	1	91	59	0	3600
R1_10_10	47364.6	46676.18	1	3600	-	-	-	-	-	3600
R1_10_5	50406.7	49928.13	1	3600	48279.2	1	34	269	0	3600
R1_10_9	49162.8	48632.73	1	3600	34575.0	1	1	0	0	3600
R1_2_1	4667.2	4667.2	1	16	4645.66	15	1731	195	0	3600
R1_2_10	3293.1	3285.31	15	3600	3150.43	1	98	1424	0	3600
R1_2_2	3919.9	3919.9	1	45	3688.7	1	99	1008	0	3600
R1_2_3	3373.9	3358.72	5	3600	2932.78	1	29	304	0	3600
R1_2_4	3047.6	3039.51	3	3600	-	-	-	-	-	3600
R1_2_5	4053.2	4053.2	3	399	3989.12	1	530	1959	0	3600
R1_2_6	3559.1	3552.78	13	3600	3328.57	1	82	1245	0	3600
R1_2_7	3141.9	3141.9	1	750	2892.17	1	33	481	0	3600
R1_2_8	2938.4	2938.4	5	2030	2699.9	1	20	249	0	3600
R1_2_9	3734.7	3734.7	3	731	3627.03	1	252	2017	0	3600
R1_4_1	10305.8	10305.8	3	338	10191.2	1	439	149	0	3600
R1_4_10	8077.8	8028.88	1	3600	7495.14	1	34	1031	0	3600
R1_4_2	8873.3	8843.47	7	3600	7759.51	1	26	609	0	3600
R1_4_3	7784.3	7698.59	9	3600	5194.35	1	1	0	0	3600
R1_4_4	7266.2	7200.12	3	3600	-	-	-	-	-	3600
R1_4_5	9184.6	9153.48	11	3600	8967.42	1	147	919	0	3600
R1_4_6	8340.4	8321.6	5	3600	7080.0	1	17	539	0	3600
R1_4_7	7599.8	7544.64	1	3600	-	-	-	-	-	3600
R1_4_8	7240.5	7161.9	1	3600	-	-	-	-	-	3600
R1_4_9	8677.5	8627.8	5	3600	8232.98	1	72	1440	0	3600
R1_6_1	21274.2	21231.91	11	3600	20773.2	1	175	441	0	3600
R1_6_10	17583.7	17344.32	3	3600	15562.7	1	14	419	0	3600
R1_6_2	18558.7	18419.8	1	3600	-	-	-	-	-	3600
R1_6_3	16874.9	16668.68	1	3600	-	-	-	-	-	3600
R1_6_4	15721.4	15538.78	1	3600	-	-	-	-	-	3600
R1_6_5	19294.9	19210.23	3	3600	18598.7	1	57	628	0	3600
R1_6_6	17763.7	17630.27	1	3600	-	-	-	-	-	3600
R1_6_7	16496.2	16300.22	1	3600	-	-	-	-	-	3600
R1_6_9	18474.1	18357.21	3	3600	17317.1	1	28	795	0	3600
R1_8_1	36345	36225.6	5	3600	35058.5	1	101	244	0	3600
R1_8_10	30918.4	30551.1	1	3600	26879.5	1	8	246	0	3600
R1_8_2	32277.6	31948.36	1	3600	-	-	-	-	-	3600
R1_8_5	33494.2	33279.06	1	3600	32087.5	1	48	641	0	3600
R1_8_6	30872.4	30460.46	1	3600	-	-	-	-	-	3600
R1_8_9	32257.3	31928.06	1	3600	29931.2	1	21	578	0	3600
R2_10_1	36881	-	-	3600	16123.8	1	21	14	0	3600
R2_2_1	3468	3468.0	1	142	3256.74	1	283	1878	0	3600
R2_2_10	2549.4	2549.4	3	1439	1322.66	1	49	136	0	3600
R2_2_2	3008.2	3008.2	1	531	941.376	1	24	72	0	3600
R2_2_3	2537.5	2537.5	1	2283	903.911	1	22	3	0	3600
R2_2_4	1928.5	1925.02	3	3600	-	-	-	-	-	3600
R2_2_5	3061.1	3061.1	1	1125	1996.68	1	92	360	0	3600
R2_2_6	2675.4	2675.4	3	2384	1023.08	1	29	74	0	3600
R2_2_7	2304.7	2298.61	1	3600	890.502	1	21	4	0	3600
R2_2_8	1842.4	1819.76	1	3600	-	-	-	-	-	3600
R2_2_9	2843.3	2843.3	1	782	1432.83	1	54	236	0	3600

Table 6 Continued

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
R2_4_1	7520.7	7520.7	1	2828	5280.74	1	80	315	0	3600
R2_4_10	5645.9	5543.96	1	3600	2240.21	1	25	41	0	3600
R2_4_2	6482.8	6374.27	1	3600	1038.87	1	1	0	0	3600
R2_4_5	6567.9	6527.42	1	3600	3156.52	1	40	157	0	3600
R2_4_6	5813.5	5643.47	1	3600	-	-	-	-	-	3600
R2_4_9	6067.8	6027.42	1	3600	2614.39	1	30	114	0	3600
R2_6_1	15145.3	-	-	3600	7881.26	1	34	38	0	3600
R2_6_5	13790.2	-	-	3600	5379.07	1	20	5	0	3600
R2_6_9	12736.8	-	-	3600	2401.37	1	1	0	0	3600
R2_8_1	24969.8	-	-	3600	12277.4	1	28	32	0	3600
R2_8_5	22801.6	-	-	3600	8014.18	1	14	0	0	3600
RC1_10_1	45790.8	45302.6	1	3600	43614.7	1	35	1879	0	3600
RC1_10_5	45028.1	44404.77	1	3600	39380.4	1	13	1050	0	3600
RC1_10_6	44903.6	44284.41	1	3600	38292.0	1	9	876	0	3600
RC1_10_7	44417.1	43820.04	1	3600	34909.7	1	1	0	0	3600
RC1_10_8	43916.5	43307.23	1	3600	-	-	-	-	-	3600
RC1_10_9	43858.1	43191.82	1	3600	-	-	-	-	-	3600
RC1_2_1	3516.9	3516.9	23	2632	3441.92	2	661	2851	0	3600
RC1_2_10	2990.5	2969.39	5	3600	2819.2	1	77	1618	0	3600
RC1_2_2	3221.6	3213.54	9	3600	3051.95	1	119	1643	0	3600
RC1_2_3	3001.4	2984.21	3	3600	2742.14	1	47	800	0	3600
RC1_2_4	2845.2	2833.72	3	3600	2514.53	1	27	678	0	3600
RC1_2_5	3325.6	3319.34	9	3600	3192.13	1	261	3554	0	3600
RC1_2_6	3300.7	3300.7	3	1028	3182.13	1	290	3885	0	3600
RC1_2_7	3177.8	3154.8	5	3600	3035.01	1	169	2991	0	3600
RC1_2_8	3060	3049.85	5	3600	2926.91	1	119	2384	0	3600
RC1_2_9	3073.3	3041.67	7	3600	2908.47	1	110	2486	0	3600
RC1_4_1	8522.9	8481.66	5	3600	8258.16	1	210	2255	0	3600
RC1_4_10	7581.2	7511.6	1	3600	6654.87	1	23	1598	0	3600
RC1_4_2	7878.2	7843.85	1	3600	7089.51	1	25	875	0	3600
RC1_4_3	7516.9	7454.57	1	3600	5768.99	1	2	0	0	3600
RC1_4_4	7292.9	7206.25	1	3600	-	-	-	-	-	3600
RC1_4_5	8152.3	8101.4	1	3600	7773.46	1	91	3143	0	3600
RC1_4_6	8148	8092.64	1	3600	7758.64	1	91	3255	0	3600
RC1_4_7	7932.5	7884.28	1	3600	7443.94	1	66	2392	0	3600
RC1_4_8	7757.2	7687.88	1	3600	7073.7	1	39	2385	0	3600
RC1_4_9	7717.7	7641.23	5	3600	7004.49	1	35	2658	0	3600
RC1_6_1	16960.1	16846.82	1	3600	16278.4	1	73	1927	0	3600
RC1_6_10	15651.3	15455.46	1	3600	12949.3	1	10	959	0	3600
RC1_6_2	15890.6	15715.75	1	3600	12229.1	1	2	0	0	3600
RC1_6_3	15181.3	14922.33	1	3600	-	-	-	-	-	3600
RC1_6_4	14753.2	14405.48	1	3600	-	-	-	-	-	3600
RC1_6_5	16536.3	16377.64	1	3600	15270.3	1	36	1999	0	3600
RC1_6_6	16473.3	16315.89	1	3600	15137.4	1	36	1974	0	3600
RC1_6_7	16055.3	15929.19	3	3600	14335.9	1	24	1972	0	3600
RC1_6_8	15891.8	15689.09	1	3600	13378.6	1	13	759	0	3600
RC1_6_9	15803.5	15611.36	1	3600	13160.9	1	13	1131	0	3600
RC1_8_1	29978.9	29723.6	1	3600	28806.8	1	63	2521	0	3600
RC1_8_10	28168.5	27725.26	1	3600	22049.3	1	1	0	0	3600
RC1_8_2	28290.1	27880.5	1	3600	-	-	-	-	-	3600
RC1_8_5	29219.9	28903.94	5	3600	27127.4	1	28	1853	0	3600

Table 6 Continued

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
RC1_8_6	29194.2	28795.58	5	3600	27133.9	1	29	2204	0	3600
RC1_8_7	28788.6	28400.17	3	3600	25532.1	1	17	1366	0	3600
RC1_8_8	28418.1	28007.4	1	3600	22049.3	1	1	0	0	3600
RC1_8_9	28347.1	27992.74	1	3600	-	-	-	-	-	3600
RC2_2_1	2797.4	2797.4	1	196	2191.22	1	538	2716	0	3600
RC2_2_10	1989.2	1954.78	1	3600	10.8883	1	20	515	0	3600
RC2_2_2	2481.6	2481.6	3	1808	589.51	1	26	155	0	3600
RC2_2_4	1854.8	1839.25	1	3600	-	-	-	-	-	3600
RC2_2_5	2491.4	2491.4	1	684	1221.36	1	157	968	0	3600
RC2_2_6	2495.1	2495.1	1	713	1391.57	1	245	1262	0	3600
RC2_2_7	2287.7	2284.42	5	3600	1063.54	1	155	1049	0	3600
RC2_2_8	2151.2	2151.2	1	2476	119.224	1	29	600	0	3600
RC2_2_9	2086.6	2059.93	1	3600	812.969	1	103	844	0	3600
RC2_4_1	6147.3	6141.13	1	3600	3802.14	1	162	1072	0	3600
RC2_4_2	5407.5	5328.56	1	3600	-	-	-	-	-	3600
RC2_4_5	5392.3	5369.86	1	3600	2118.76	1	50	287	0	3600
RC2_4_6	5324.6	5253.47	1	3600	2211.67	1	59	353	0	3600
RC2_4_7	4987.8	4848.73	1	3600	1099.37	1	1	0	0	3600
RC2_4_8	4693.3	4126.88	1	3600	-	-	-	-	-	3600
RC2_4_9	4510.4	2898.74	1	3600	-	-	-	-	-	3600
RC2_6_1	11966.1	-	-	3600	4797.26	1	49	344	0	3600
RC2_8_1	19201.3	-	-	3600	6242.12	1	17	211	0	3600

Appendix M: Graph Multicoloring Problem Results

In Table 7 we compare column elimination with the branch-and-price algorithm from Gualandi and Malucelli (2012) on graph multi-coloring problems. For each instance, we list the number of nodes (n) and edges (m) in the graph, and value of the maximum weighted clique (ω) used as a lower bound for preprocessing. For the branch-and-price algorithm (GM) we report the lower bound (LB), upper bound (UB), and the solution time (Time (s)), taken from the original paper. For column elimination, we report the lower bound (LB), the number of column elimination iterations (CEIt), the number of refinements (CR), and the solution time (Time (s)).

Table 7 A comparison of the branch-and-price algorithm from Gualandi and Malucelli (2012) (GM) and column elimination for solving COG instances created by Gualandi and Malucelli (2012).

Instance				GM			Column Elimination				
Name	n	m	ω	LB	UB	Time (s)	LB	UB	CEIt	CR	Time (s)
COG-10teams	3200	124480	73	-	-	3600	71	-	24	3093	3600
COG-air04	17808	2121648	377	377	377	1.8	377	377	2	0	8
COG-air05	14390	2527253	413	-	-	3600	-	-	0	0	996
COG-atlanta-ip	8124	9250	15	15	15	1844	15	-	173	4076	3600
COG-cap6000	11992	12103	14	14	14	304	14	14	2	0	1
COG-ds	15252	2057486	1	500	500	6.5	-	-	-	-	3600
COG-gesa2-o	192	144	12	12	13	3600	12	**12**	2	0	0
COG-misc07	410	2928	36	36	39	3600	36	**36**	103	378	151
COG-mkc	10394	154870	169	169	169	0.1	169	169	2	0	1
COG-mod011	192	336	12	12	13	3600	11	-	53	1716	3600
COG-mzzv11	19942	257012	101	101	101	0.1	101	101	2	0	6
COG-mzzv42z	18806	225687	91	91	91	0.1	91	91	2	0	5
COG-net12	3202	4835	17	17	17	1301	17	-	104	2600	3600
COG-nsrand-ipx	13240	69510	30	-	-	3600	30	**30**	2	0	5
COG-opt1217	1536	6528	26	-	-	3600	26	**26**	2	0	14
COG-rd-rplusc-21	904	11785	109	109	109	0.0	-	-	-	-	3600
COG-rout	560	2940	30	30	32	3600	30	**30**	2	0	0
COG-swath	12480	958000	317	-	-	3600	317	-	10	134	3600

Appendix N: Pickup-and-Delivery Problem with Time Windows Results

In Table 7 we compare column elimination with the dual ascent method from Baldacci et al. (2011a) (BBM) and VRPSolver on pickup-and-delivery problems with time windows (PDPTW). For each instance, we list the best known upper bound (UB). For BBM we report the lower bound (LB), upper bound (UB), and the solution time (Time (s)), taken from the original paper. For VRPSolver we report the lower bound (LB) and the solution time (Time (s)). For column elimination, we report the lower bound (LB), the number of column elimination iterations (CEIt) and column elimination with subgradient descent iterations (CESIt), the number of refinements (CR), and the solution time (Time (s)).

Table 9 shows the performance of column elimination for solving PDPTW instances by Li and Lim (2001) that have not yet been reported on by an exact solver.

Table 8 **Comparing the dual ascent method from Baldacci et al. (2011a) (BBM), VRPSolver, and column elimination for solving a selection of PDPTW instances by Li and Lim (2001) with 200 locations.**

Instance		BBM			VRPSolver		Column Elimination				
Name	UB	LB	UB	Time (s)	LB	Time (s)	LB	CEIt	CESIt	CR	Time (s)
LC1_2_1	2704.6	2704.6	2704.6	3.3	-	3600	2704.57	8	30	64	3600
LC1_2_10	2741.6	2741.6	2741.6	137.1	-	3600	2488.16	1	1425	24479	3600
LC1_2_2	2764.6	2764.6	2764.6	21.5	-	3600	2764.56	27	220	2302	3600
LC1_2_3	2772.2	2772.2	2772.2	114.9	-	3600	2694.04	1	678	7783	3600
LC1_2_4	2661.4	2395.8	2661.4	454.2	-	3600	2340.5	1	418	8118	3600
LC1_2_5	2702.0	2702.0	2702.0	4.8	-	3600	2702.05	8	40	234	10
LC1_2_6	2701.0	2701.0	2701.0	7.4	-	3600	2701.04	9	50	353	17
LC1_2_7	2701.0	2701.0	2701.0	7.7	-	3600	2701.04	8	50	355	21
LC1_2_8	2689.8	2689.8	2689.8	16.0	-	3600	2683.6	778	79708	4285	3600
LC1_2_9	2724.2	2724.2	2724.2	55.3	-	3600	2684.86	6	1942	26833	3600
LR1_2_1	4819.1	4819.1	4819.1	1.6	-	3600	4819.12	8	80	662	8
LR1_2_10	3386.3	3386.3	3386.3	1376.7	-	3600	2706.01	1	789	7763	3600
LR1_2_2	4093.1	4093.1	4093.1	20.6	-	3600	4055.38	1	400	3973	3600
LR1_2_3	3486.8	3486.8	3486.8	3690.8	-	3600	-	-	-	-	3600
LR1_2_4	2830.7	2341.8	2830.7	1809.6	-	3600	-	-	-	-	3600
LR1_2_5	4221.6	4221.6	4221.6	2.6	-	3600	4192.38	73	7758	16207	3600
LR1_2_6	3763.0	3763.0	3763.0	180.9	-	3600	3356.97	1	354	3966	3600
LR1_2_7	3112.9	2761.8	3112.9	1320.4	-	3600	1830.31	1	95	707	3600
LR1_2_8	2645.5	2150.8	2645.5	566.9	-	3600	-	-	-	-	3600
LR1_2_9	3953.5	3953.3	3953.5	15.4	-	3600	3779.64	2	2006	21428	3600
LRC1_2_1	3606.1	3606.1	3606.1	3.1	-	3600	3594.24	67	7350	5221	3600
LRC1_2_10	2837.5	2335.5	2837.5	217.4	-	3600	2181.99	1	1001	10503	3600
LRC1_2_2	3292.4	3292.4	3292.4	322.3	-	3600	3020.23	1	850	9601	3600
LRC1_2_3	3079.5	2497.8	3079.5	304.3	-	3600	-	-	-	-	3600
LRC1_2_4	2525.8	1981.0	2525.8	188.2	-	3600	-	-	-	-	3600
LRC1_2_5	3715.8	3715.8	3715.8	42.1	-	3600	3280.25	1	2845	41795	3600
LRC1_2_6	3360.9	3360.9	3360.9	7.0	-	3600	3205.94	2	2907	35316	3600
LRC1_2_7	3317.7	3317.7	3317.7	408.2	-	3600	2795.52	1	2310	30413	3600
LRC1_2_8	3086.5	3086.5	3086.5	1562.7	-	3600	2386.28	2	1636	20021	3600
LRC1_2_9	3053.8	3053.8	3053.8	1757.2	-	3600	2375.2	4	1479	19662	3600
LC1_10_1	42488.66	42488.7	42488.7	79.5	-	3600	42488.7	8	70	951	149
LC1_10_5	42477.4	42477.4	42477.4	118.7	-	3600	42477.4	8	210	4843	2232
LR1_10_1	56744.91	56744.9	56744.9	233.1	-	3600	55200.4	1	639	29085	3600
LR1_10_5	59053.68	52536.3	52901.3	4068.8	-	3600	43578.9	1	163	7065	3600
LRC1_10_1	49111.78	48398.8	48666.5	2533.3	-	3600	38724.2	1	186	9161	3600
LRC1_10_5	50323.04	38177.8	49287.1	1650.3	-	3600	2112.59	1	22	3865	3600

Table 9 The performance of column elimination for solving PDPTW instances by Li and Lim (2001) that have not yet been reported on by an exact solver.

Instance		Column Elimination				
Name	UB	LB	CEIt	CESIt	CR	Time (s)
LC1_10_6	42838.39	42512.1	1	269	11426	3600
LC1_10_7	42854.99	42482.7	1	360	14140	3600
LC1_10_8	42949.56	36753.0	1	99	5731	3600
LC1_10_9	43564.01	10545.2	1	3	296	3600
LC1_4_1	7152.06	**7152.06**	8	30	152	15
LC1_4_10	7850.22	6048.13	1	265	8047	3600
LC1_4_2	8007.79	6849.74	1	131	2031	3600
LC1_4_3	8678.23	5839.12	1	81	1394	3600
LC1_4_4	6451.68	4955.73	1	52	798	3600
LC1_4_5	7150.0	**7150.0**	8	50	435	54
LC1_4_6	7154.02	**7154.02**	16	80	1218	1349
LC1_4_7	7149.43	**7149.43**	11	90	1583	712
LC1_4_8	8305.42	7077.22	6	1171	24073	3600
LC1_4_9	7451.2	6242.41	1	230	7154	3600
LC1_6_1	14095.64	**14095.6**	9	40	328	71
LC1_6_2	15048.16	12851.8	1	82	2113	3600
LC1_6_3	14874.68	10270.8	1	44	856	3600
LC1_6_5	14086.3	**14086.3**	8	90	1307	358
LC1_6_6	14090.79	**14090.8**	8	150	2969	735
LC1_6_7	14083.76	**14083.8**	9	120	2791	2310
LC1_6_8	14880.7	12983.3	1	114	4320	3600
LC1_6_9	15207.58	11524.3	1	79	2442	3600
LC1_8_1	25184.38	**25184.4**	8	50	591	92
LC1_8_2	29735.85	5436.04	1	2	186	3600
LC1_8_5	25211.22	**25211.2**	8	100	2398	828
LC1_8_6	25164.25	**25164.3**	12	210	4882	3009
LC1_8_7	25158.38	25158.3	6	190	6585	3600
LC1_8_8	26688.17	21526.2	1	59	2391	3600
LC1_8_9	26975.61	17537.6	1	40	828	3600
LC2_10_1	16879.24	16834.4	2	90	1269	3600
LC2_10_10	17043.64	13641.0	1	72	1388	3600
LC2_10_5	17137.53	15746.7	1	232	6644	3600
LC2_10_6	19387.75	15088.3	1	142	3760	3600
LC2_10_7	18389.37	14776.0	1	96	2005	3600
LC2_10_8	17015.03	14450.1	1	102	2355	3600
LC2_10_9	18225.3	10031.1	1	22	40	3600
LC2_2_1	1931.44	**1931.44**	16	100	478	86
LC2_2_10	1817.45	1706.59	1	796	5085	3600
LC2_2_2	1881.4	1854.18	2	669	3844	3600
LC2_2_3	1844.33	1701.93	1	282	2046	3600
LC2_2_4	1767.12	1517.13	1	143	705	3600
LC2_2_5	1891.21	1877.31	10	1885	13262	3600
LC2_2_6	1857.78	1804.46	1	1194	8664	3600
LC2_2_7	1850.13	1806.89	2	907	6338	3600
LC2_2_8	1824.34	1754.23	1	1009	7019	3600
LC2_2_9	1854.21	1754.86	1	817	5696	3600
LC2_4_1	4116.33	**4116.33**	10	150	1181	563
LC2_4_10	3828.44	3412.01	1	182	1865	3600

Table 9 **Continued**

Instance		Column Elimination				
Name	UB	LB	CEIt	CESIt	CR	Time (s)
LC2_4_2	4144.29	3916.04	1	245	3061	3600
LC2_4_3	4401.08	3128.95	1	81	533	3600
LC2_4_4	3743.95	1793.21	1	27	43	3600
LC2_4_5	4030.63	3884.4	1	516	6415	3600
LC2_4_6	3900.29	3687.84	1	255	3015	3600
LC2_4_7	3962.51	3694.87	1	284	3597	3600
LC2_4_8	3844.45	3586.93	1	284	3394	3600
LC2_4_9	4188.93	3527.43	1	181	1814	3600
LC2_6_1	7977.98	7876.53	1	556	5794	3600
LC2_6_10	7946.6	6371.09	1	103	1450	3600
LC2_6_2	9900.48	7331.11	1	159	2502	3600
LC2_6_3	8512.94	5518.71	1	45	217	3600
LC2_6_5	9051.53	7344.22	1	446	8306	3600
LC2_6_6	8775.55	7037.1	1	242	4162	3600
LC2_6_7	9376.58	6875.76	1	117	1800	3600
LC2_6_8	7579.63	6767.95	1	167	2550	3600
LC2_6_9	8714.22	6763.4	1	135	1968	3600
LC2_8_1	11687.06	11649.6	9	120	1372	3600
LC2_8_10	12226.42	9442.24	1	73	965	3600
LC2_8_2	13713.13	9562.29	1	45	228	3600
LC2_8_5	12298.33	11068.8	1	264	5907	3600
LC2_8_6	12645.71	10566.1	1	179	3769	3600
LC2_8_7	14041.47	10317.8	1	100	1590	3600
LC2_8_8	12924.1	10045.1	1	105	1653	3600
LC2_8_9	11629.41	9919.23	1	87	1354	3600
LR1_10_9	54582.25	35187.2	1	79	2017	3600
LR1_4_1	10639.75	**10639.8**	9	530	3906	437
LR1_4_10	8192.65	126.225	1	39	2340	3600
LR1_4_5	11374.06	9381.73	4	1274	21471	3600
LR1_4_9	9859.47	7648.39	1	429	8360	3600
LR1_6_1	22821.65	22452.5	7	1194	17635	3600
LR1_6_10	19298.25	10244.8	1	92	1551	3600
LR1_6_5	23623.52	18469.2	1	370	11561	3600
LR1_6_9	21835.87	14816.1	1	158	3374	3600
LR1_8_1	39291.32	38716.1	1	949	25746	3600
LR1_8_5	39809.68	30320.6	1	210	8448	3600
LR1_8_9	38499.0	24194.6	1	113	2555	3600
LR2_2_1	4073.1	3346.21	1	296	1633	3600
LR2_2_10	3254.83	1279.46	1	60	44	3600
LR2_2_2	3796.0	908.641	1	40	19	3600
LR2_2_3	3098.36	197.901	1	1	0	3600
LR2_2_5	3438.39	1922.72	1	107	261	3600
LR2_2_6	4457.95	1074.38	1	47	10	3600
LR2_2_9	3922.11	1396.92	1	70	134	3600
LR2_4_1	9726.88	4821.87	1	82	159	3600
LR2_4_10	7596.62	1937.65	1	32	0	3600
LR2_4_5	9894.46	2861.37	1	53	55	3600
LR2_4_9	7926.07	2605.59	1	44	33	3600
LR2_6_1	21759.33	7051.15	1	47	27	3600
LR2_6_5	22625.89	1248.95	1	1	0	3600

Table 9 Continued

Instance		Column Elimination				
Name	UB	LB	CEIt	CESIt	CR	Time (s)
LR2_6_9	18259.2	1260.05	1	2	0	3600
LR2_8_1	41873.12	10640.0	1	38	1	3600
LRC1_10_6	45115.22	28409.1	1	128	6034	3600
LRC1_4_1	9124.52	8786.89	9	2219	27012	3600
LRC1_4_5	8847.4	7362.81	1	1208	32476	3600
LRC1_4_6	8394.47	7055.52	1	1315	33591	3600
LRC1_4_7	8037.87	5980.69	1	787	18678	3600
LRC1_4_8	7930.15	5244.7	1	540	11088	3600
LRC1_4_9	8004.24	5290.96	1	678	14613	3600
LRC1_6_1	18288.9	17196.9	1	993	27370	3600
LRC1_6_5	17463.94	13599.9	1	470	15706	3600
LRC1_6_6	19025.36	12819.5	1	473	15816	3600
LRC1_6_7	15914.85	11060.8	1	277	8521	3600
LRC1_8_1	32252.28	29266.0	1	477	21035	3600
LRC1_8_5	31169.16	20067.0	1	180	7759	3600
LRC1_8_6	28961.66	3690.2	1	53	6481	3600
LRC2_2_1	3595.18	2208.29	1	555	2737	3600
LRC2_2_10	2437.88	688.411	1	74	388	3600
LRC2_2_2	3158.25	568.591	1	35	74	3600
LRC2_2_5	2776.93	1378.96	1	206	806	3600
LRC2_2_6	2707.96	1290.34	1	243	1037	3600
LRC2_2_7	3010.68	1038.16	1	133	649	3600
LRC2_2_8	2399.89	878.886	1	105	460	3600
LRC2_2_9	2208.49	932.522	1	115	481	3600
LRC2_4_1	9738.95	3925.03	1	189	1034	3600
LRC2_4_10	5395.71	591.091	1	1	0	3600
LRC2_4_5	7309.54	2224.78	1	76	255	3600
LRC2_4_6	6337.08	2343.81	1	88	303	3600
LRC2_4_7	6292.23	1618.1	1	36	81	3600
LRC2_4_9	6272.92	1585.12	1	39	90	3600
LRC2_6_1	14578.92	4764.83	1	73	339	3600
LRC2_8_1	23074.22	2250.61	1	1	0	3600

Appendix O: Sequential Ordering Problem Results

In Table 10 we compare column elimination with the tree-search solver from (Libralesso et al. 2020) (CATS) on sequential ordering problem (SOP) instances from TSPLIB. For each instance, we list the best known upper bound (UB). For CATS we report the lower bound (LB) and the solution time (Time (s)), taken from the original paper. For column elimination, we report the lower bound (LB), the number of column elimination iterations (CEIt) and column elimination with subgradient descent iterations (CESIt), the number of refinements (CR), and the solution time (Time (s)).

Table 10 **Comparing the performance of the tree-search solver from (Libralesso et al. 2020) (CATS) and column elimination for solving SOP instances from TSPLIB.**

Instance		CATS		Column Elimination				
Name	UB	LB	Time (s)	LB	CEIt	CESIt	CR	Time (s)
ESC07	2125	2125	0	2125.0	24	1	32	0
ESC11	2075	2075	0	2075.0	12	1	26	0
ESC12	1675	1675	0	1675.0	180	1	1714	3
ESC25	1681	818	3600	1681.0	320	1	6177	274
ESC47	1288	315	3600	998.1	126	60	2526	3600
ESC63	62	4	3600	54.9	198	20673	20476	3600
ESC78	18230	850	3600	10821.4	236	30070	29835	3600
br17.10	55	55	59	55.0	1680	1	12355	1145
br17.12	55	55	25	55.0	1543	1	12864	1173
ft53.1	7531	1036	3600	6825.8	96	13800	13705	3600
ft53.2	8026	1136	3600	6376.1	148	17088	16941	3600
ft53.3	10262	2119	3600	7086.2	535	57807	57273	3600
ft53.4	14425	14425	241	9246.8	838	88421	87584	3600
ft70.1	39313	3094	3600	38345.7	87	9216	9130	3600
ft70.2	40419	2877	3600	38543.8	110	11592	11483	3600
ft70.3	42535	4490	3600	39380.9	338	35000	34663	3600
ft70.4	53530	20269	3600	43999.1	480	49384	48905	3600
kro124p.1	39420	2467	3600	34870.7	30	4380	4351	3600
kro124p.2	41336	3116	3600	34971.7	42	5756	5715	3600
kro124p.3	49499	4065	3600	36040.5	174	19699	19526	3600
kro124p.4	76103	7150	3600	42348.2	229	26183	25955	3600
p43.1	28140	215	3600	4545.2	79	40511	40433	3600
p43.2	28480	295	3600	4649.4	91	47493	47403	3600
p43.3	28835	585	3600	4750.2	207	64027	63821	3600
p43.4	83005	83005	183	6541.4	901	123460	122560	3600
prob.100	1163	149	3600	793.1	1	4160	4160	3600
prob.42	243	82	3600	225.8	131	70	2219	3600
rbg048a	351	63	3600	322.3	201	23482	23282	3600
rbg050c	467	70	3600	428.0	386	43948	43563	3600
rbg109a	1038	117	3600	927.9	107	17006	16900	3600
rbg150a	1750	123	3600	1132.0	1	7376	7376	3600
rbg174a	2033	471	3600	1251.6	1	5234	5233	3600
rbg253a	2950	140	3600	1267.3	1	1292	1292	3600
rbg323a	3140	74	3600	1262.1	1	726	726	3600
rbg341a	2568	67	3600	67.0	1	637	636	3600
rbg358a	2545	101	3600	77.4	1	451	451	3600
rbg378a	2816	66	3600	51.6	1	446	446	3600
ry48p.1	15805	2856	3600	14220.1	135	14401	14267	3600
ry48p.2	16666	3779	3600	14383.6	184	19446	19263	3600
ry48p.3	19894	5268	3600	15094.8	487	50826	50340	3600
ry48p.4	31446	31446	57	22176.1	1148	120079	118932	3600

Appendix P: Graph Coloring Problem Results

In Table 10 we compare column elimination with the branch-and-price algorithm by (Held et al. 2011) (Held et al.) on graph coloring instances from the DIMACS benchmark set. For each instance, we list the number of nodes (n) and edges (m), and the best known solution (BKS). For both methods we report the lower bound (LB) and the solution time (Time (s)). Both methods are run on the same server.

Table 11 A comparison of the branch-and-price algorithm by (Held et al. 2011) and column elimination on graph coloring problem instances from the DIMACS benchmark set.

Instance				Held et al.		Column Elimination	
Name	n	m	BKS	LB	Time (s)	LB	Time (s)
1-FullIns_3	30	100	4	4	0	4	0
1-FullIns_4	93	593	5	4	3600	5	5
1-FullIns_5	282	3247	6	5	3600	6	3469
1-Insertions_4	67	232	5	4	3600	3	3600
1-Insertions_5	202	1227	6	5	3600	3	3600
1-Insertions_6	607	6337	7	6	3600	3	3600
2-FullIns_3	52	201	5	5	0	5	0
2-FullIns_4	212	1621	6	5	3600	6	3
2-FullIns_5	852	12201	7	6	3600	5	3600
2-Insertions_3	37	72	4	4	206	3	3600
2-Insertions_4	149	541	5	4	3600	3	3600
2-Insertions_5	597	3936	6	5	3600	3	3600
3-FullIns_3	80	346	6	6	0	6	0
3-FullIns_4	405	3524	7	6	3600	7	36
3-FullIns_5	2030	33751	8	6	3600	5	3600
3-Insertions_3	56	110	4	3	3600	3	3600
3-Insertions_4	281	1046	5	4	3600	3	3600
3-Insertions_5	1406	9695	6	3	3600	2	3600
4-FullIns_3	114	541	7	7	0	7	0
4-FullIns_4	690	6650	8	7	3600	8	65
4-FullIns_5	4146	77305	9	7	3600	6	3600
4-Insertions_3	79	156	4	3	3600	3	3600
4-Insertions_4	475	1795	5	4	3600	3	3600
5-FullIns_3	154	792	8	8	0	8	0
5-FullIns_4	1085	11395	9	8	3600	9	277
C2000.5	2000	999836	145	1	3600	12	3600
C2000.9	2000	1799532	400	1	3600	103	3600
C4000.5	4000	4000268	259	1	3600	1	3600
DSJC1000.1	1000	49629	20	1	3600	5	3600
DSJC1000.5	1000	249826	82	1	3600	14	3600
DSJC1000.9	1000	449449	222	217	3600	1	3600
DSJC125.1	125	736	5	5	1	5	31
DSJC125.5	125	3891	17	17	129	13	3600
DSJC125.9	125	6961	44	44	1	43	3600
DSJC250.1	250	3218	8	7	3600	5	3600
DSJC250.5	250	15668	28	27	3600	14	3600
DSJC250.9	250	27897	72	72	1757	71	3600

Table 11 **Continued**

Instance				Held et al.		Column Elimination	
Name	n	m	BKS	LB	Time (s)	LB	Time (s)
DSJC500.1	500	12458	12	1	3600	5	3600
DSJC500.5	500	62624	47	44	3600	14	3600
DSJC500.9	500	224874	126	125	3600	118	3600
DSJR500.1	500	3555	12	12	0	12	32
DSJR500.1c	500	121275	85	85	9	85	66
DSJR500.5	500	58862	122	122	29	119	3600
abb313GPIA	1557	65390	9	8	3600	7	3600
anna	138	986	11	11	0	11	0
ash331GPIA	662	4185	4	4	1	3	3600
ash608GPIA	1216	7844	4	4	856	2	3600
ash958GPIA	1916	12506	4	1	3600	3	3600
david	87	812	11	11	0	11	0
flat1000_50_0	1000	245000	50	50	1658	13	3600
flat1000_60_0	1000	245830	60	1	3600	14	3600
flat1000_76_0	1000	246708	81	1	3600	14	3600
flat300_20_0	300	21375	20	20	4	13	3600
flat300_26_0	300	21633	26	26	14	13	3600
flat300_28_0	300	21695	28	28	9	13	3600
fpsol2.i.1	496	11654	65	65	0	65	6
fpsol2.i.2	451	8691	30	30	0	30	37
fpsol2.i.3	425	8688	30	30	0	30	37
games120	120	1276	9	9	0	9	34
homer	561	3258	13	13	0	13	1
huck	74	602	11	11	0	11	0
inithx.i.1	864	18707	54	54	1	54	46
inithx.i.2	645	13979	31	31	0	31	64
inithx.i.3	621	13969	31	31	0	31	65
jean	80	508	10	10	0	10	0
latin_square_10	900	307350	98	91	3600	90	3600
le450_15a	450	8168	15	15	8	13	3600
le450_15b	450	8169	15	15	9	15	31
le450_15c	450	16680	15	15	166	11	3600
le450_15d	450	16750	15	1	3600	11	3600
le450_25a	450	8260	25	25	1	20	3600
le450_25b	450	8263	25	25	1	25	34
le450_25c	450	17343	25	25	13	22	3600
le450_25d	450	17425	25	25	12	19	3600
le450_5a	450	5714	5	5	677	5	32
le450_5b	450	5734	5	5	706	5	32
le450_5c	450	9803	5	5	131	5	37
le450_5d	450	9757	5	5	65	5	30
miles1000	128	6432	42	42	0	42	0
miles1500	128	10396	73	73	0	73	0
miles250	128	774	8	8	0	8	0
miles500	128	2340	20	20	0	20	0
miles750	128	4226	31	31	0	31	0
mug100_1	100	166	4	4	0	1	3600
mug100_25	100	166	4	4	0	1	3600
mug88_1	88	146	4	4	0	3	3600
mug88_25	88	146	4	4	0	1	3600

Table 11 Continued

Instance				Held et al.		Column Elimination	
Name	n	m	BKS	LB	Time (s)	LB	Time (s)
mulsol.i.1	197	3925	49	49	0	49	0
mulsol.i.2	188	3885	31	31	0	31	5
mulsol.i.3	184	3916	31	31	0	31	5
mulsol.i.4	185	3946	31	31	0	31	8
mulsol.i.5	186	3973	31	31	0	31	13
myciel3	11	20	4	4	0	4	0
myciel4	23	71	5	5	4	5	2
myciel5	47	236	6	5	3600	4	3600
myciel6	95	755	7	6	3600	4	3600
myciel7	191	2360	8	7	3600	3	3600
qg.order100	10000	990000	100	1	3600	1	3600
qg.order30	900	26100	30	30	78	30	49
qg.order40	1600	62400	40	40	480	40	90
qg.order60	3600	212400	60	1	3600	60	2547
queen10_10	100	2940	11	11	109	10	3600
queen11_11	121	3960	11	11	0	11	30
queen12_12	144	5192	12	12	0	12	30
queen13_13	169	6656	13	13	1	13	33
queen14_14	196	8372	14	14	1	14	34
queen15_15	225	10360	15	15	2	15	30
queen16_16	256	12640	16	16	4	16	30
queen5_5	25	320	5	5	0	5	0
queen6_6	36	580	7	7	0	7	0
queen7_7	49	952	7	7	0	7	0
queen8_12	96	2736	12	12	0	12	31
queen8_8	64	1456	9	9	0	9	32
queen9_9	81	2112	10	10	3	9	3600
r1000.1	1000	14378	20	20	0	20	44
r1000.1c	1000	485090	97	96	3600	85	3600
r1000.5	1000	238267	234	234	380	227	3600
r125.1	125	209	5	5	0	5	0
r125.1c	125	7501	46	46	0	46	0
r125.5	125	3838	36	36	0	36	57
r250.1	250	867	8	8	0	8	0
r250.1c	250	30227	64	64	0	64	3
r250.5	250	14849	65	65	2	64	3600
school1	385	19095	14	14	53	1	3600
school1_nsh	352	14612	14	14	18	1	3600
wap01a	2368	110871	41	1	3600	39	3600
wap02a	2464	111742	41	1	3600	38	3600
wap03a	4730	286722	44	1	3600	38	3600
wap04a	5231	294902	42	1	3600	1	3600
wap05a	905	43081	50	50	4	40	3600
wap06a	947	43571	40	40	106	35	3600
wap07a	1809	103368	41	1	3600	38	3600
wap08a	1870	104176	41	1	3600	34	3600
will199GPIA	701	7065	7	7	1	6	3600
zeroin.i.1	211	4100	49	49	0	49	0
zeroin.i.2	211	3541	30	30	0	30	0
zeroin.i.3	206	3540	30	30	0	30	0