Carnegie Mellon University
Tepper School of Business

Doctoral Dissertation

Column Elimination: Iterative Refinement for Solving Arc Flow Formulations

Anthony Karahalios

April 2025

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Algorithms, Combinatorics and Optimization.

Dissertation Committee:
Merve Bodur
Ricardo Fukasawa
John Hooker
Fatma Kılınç-Karzan
Willem-Jan van Hoeve (Chair)

Acknowledgments

This work is partially supported by Office of Naval Research Grant No. N00014-21-1-2240 and National Science Foundation Award #1918102. This material is also based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE1745016, DGE2140739. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Chapter 1

Introduction

An optimization problem is to find the best decisions among a set of alternatives. There are constraints that determine which decisions are feasible, and there is a metric that determines the value of the decisions. This type of problem arises in many business settings. For example, in the transportation sector, a shipping company wants to deliver packages as efficiently as possible within the constraints of its budget, assets, and workforce.

An optimization model is a mathematical description of an optimization problem. It has three parts: decision variables, constraints, and an objective function. Decision variables are placeholders for assignments of values that represent decisions. A constraint is a function that takes an assignment of values to the decision variables as input and outputs whether or not the assignment is feasible. An objective function takes an assignment of values to the decision variables as input and outputs a real number that represents its value.

An optimization algorithm is a method to solve an optimization model. There are exact optimization algorithms and heuristic optimization algorithms. An exact algorithm aims not only to find the best solution, but also to prove that no better solution exists. A heuristic algorithm aims to find a good solution to the model without proving how good it is. An optimization algorithm is usually designed to solve a particular form of an optimization model. For example, an optimization algorithm might solve optimization models that have linear objective functions and linear constraints.

1.1 Motivation

The purpose of this work is to present a dissertation on column elimination, which is a novel framework for modeling and exactly solving a particular form of optimization problem. The optimization problem is described by sequences of decisions: Given a set of feasible sequences of decisions, choose the best subset of these feasible sequences that collectively meet some constraints. For example, consider a delivery company with a fleet of vehicles. Assume that all delivery vehicles start at a common depot where the packages are stored. The set of feasible sequences of decisions is equivalent to the set of routes that a delivery vehicle can complete. The problem is to assign each vehicle a feasible route so that the vehicles collectively visit all the locations that require packages while minimizing the total cost of the routes.

The difficulty in solving the problem with an exact algorithm is that the algorithm must prove

the optimality of a solution by showing that there is no better solution. Proving optimality requires reasoning that all other feasible solutions are not better than the given solution. This can be challenging for a few reasons. First, the number of solutions can be huge. This is because the number of feasible sequences can be large and a solution is a collection of feasible sequences. Second, the description of the set of solutions may be implicitly given by complicated constraints. Third, the set of solutions and their objective values may lack a nice structure which prohibits arguments to prove many solutions to be suboptimal at the same time.

The exact algorithm that gives the best performance for many of these problems uses a decomposition approach. The problem is decomposed into two subproblems. The first subproblem restricts solutions to use only a subset of the feasible sequences, which can greatly simplify the problem. The second subproblem is to find a feasible sequence to add to this subset that admits a better solution to the first subproblem. The algorithm iterates between solving these two subproblems, and it terminates when there is no solution to the second subproblem that would improve the solution to the first subproblem. Although this method performs well for solving many problems, there are still problems that it cannot solve efficiently.

The key idea of column elimination is to use relaxations instead of decompositions. A relaxation of an optimization model removes or relaxes one or more of the constraints, allowing some infeasible solutions to be considered feasible. Somewhat surprisingly, a relaxation of the model can have fewer decision variables, making it easier to solve. However, the optimal solution to a relaxation may be infeasible. So, column elimination starts with an initial relaxation and iterates between solving a relaxation and improving the relaxation to remove any infeasible sequences found in the optimal solution. Ideally, an easier-to-solve relaxation has an optimal solution that is feasible to the original model, which implies that the solution is also an optimal solution to the original model.

Initial research on column elimination shows promising results. A problem-specific implementation of column elimination to solve the well-known vertex coloring problem achieves reasonably competitive performance with the state-of-the-art approach based on column generation. Similarly, a problem-specific implementation of column elimination to solve a truck-drone routing problem solved some instances for the first time. Since then, more research has improved and generalized the framework. This dissertation gives an overview of column elimination and focuses on the latest research.

The name column elimination comes from the name of the exact algorithm that uses a decomposition approach. That method is known as column generation because the second subproblem generates sequences to add to the first subproblem, and the sequences are referred to as columns. Meanwhile, column elimination starts with a relaxation that considers a superset of the feasible sequences and removes infeasible sequences that appear in the optimal solution to the current relaxation, equivalent to eliminating columns.

1.2 Outline and Contributions

The dissertation is organized into chapters that are based on two published works (Karahalios and van Hoeve 2022, 2023b), one submitted work (Karahalios and van Hoeve 2024), and two works in progress. The works include many computational studies for which the code is available in two repositories: (`https://github.com/amkarahalios/dd_graph_color`) and (`https://github.com/amkarahalios/dd_vrptw`).

Chapter 2 gives the necessary background information, notation, and definitions that are used throughout the dissertation. The background information is mainly about discrete optimization

methodologies including dynamic programming, integer programming, and decision diagrams. Some exact methods are mentioned that have similarities with column elimination.

Chapters 3 and 4 describe the column elimination framework. Chapter 3 gives a concrete example of using column elimination to solve the capacitated vehicle routing problem. This chapter includes the development of a Lagrangian method within column elimination, a method to incorporate cutting planes, and the use of variable fixing. Chapter 4 introduces a general version of column elimination. This chapter includes a generic conflict refinement algorithm based on a new definition of a relaxed dynamic program, a method for embedding column elimination in branch-and-bound, and experimental results on three more problems which include closing several open instances.

Chapters 5, 6, and 7 offer improvements to column elimination. Chapter 5 introduces primal heuristics for exact algorithms that solve arc flow formulations. The primal heuristics are based on the well-known method of large neighborhood search. Experimental results are shown on vehicle routing problems. Chapter 6 shows how cutting planes from path flow formulations can be translated into cutting planes for arc flow formulations. The refinement method from column elimination is used to allow the cutting planes to be expressed in the arc flow formulation. Experimental results show how the addition of these cuts improves the performance of column elimination on capacitated vehicle routing problems. Chapter 7 describes how a portfolio of variable orderings can be used to improve column elimination for solving the graph coloring problem.

Chapter 8 is a conclusion of the dissertation. This chapter includes remarks about how column elimination differs from existing methods, thoughts on the types of problems for which column elimination can perform well, and directions for future work.

Chapter 2

Discrete Optimization Methodologies

This chapter introduces definitions, notation, and background information on discrete optimization methodologies that are necessary to understand column elimination. The discrete optimization methodologies are dynamic programming, integer programming, and decision diagrams. Exact algorithms are described which are similar to column elimination.

2.1 Dynamic Programming

Dynamic programming is a method that uses a state-based system and a recursive function to solve multistage decision problems (Bellman 1957). This work focuses on dynamic programming to solve discrete deterministic finite-stage problems. In particular, consider the *minimization problem of a discrete decision process* as defined in Karp and Held (1967). Let U be a universe of elements, $\mathcal{S}$ be a set of ordered sequences of elements, each with arbitrary but finite length, and $f : \mathcal{S} \to \mathbb{R}$ be a cost function over $\mathcal{S}$. The problem is the following:

$$\min_{x \in \mathcal{S}} f(x) \tag{2.1}$$

A *dynamic program* is a model for the minimization problem of a discrete decision process that uses states and transitions to encode the set of sequences of decisions and their costs. An important attribute of the model is that the cost of a sequence can be determined recursively. This work further assumes an additive cost structure, meaning that each transition is associated with a cost, and the cost of a sequence is equal to the sum of the costs of its transitions. Formally, a dynamic program $P = (S, h, c)$ is defined by a set of states S, a set of transitions U, an initial state $r \in S$, a terminal state $t \in S$, a state transition function $h : (S \times U) \to S$ and a cost function $c : (S \times U) \to \mathbb{R}$. A *solution*, often referred to as a policy, 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 $h(s_k, u_k) = t$. The *cost* of a solution is $\sum_{i=1}^{k} c((s_i, u_i))$. For any f and $\mathcal{S}$, there exists a dynamic program such that the set of solutions and their costs is equivalent to the set of sequences $\mathcal{S}$ with costs f (Karp and Held 1967). The minimization problem is equivalent to solving $Z(r)$ where $Z : S \to \mathbb{R}$ is a

recursive value function with a base case $Z(t) = 0$ and the following form otherwise:

$$Z(s) = \min\{Z(s') + c(s', u) : h(s', u) = s\} \tag{2.2}$$

A *state-transition graph* of a dynamic program is its network representation. It 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 $P = (S, h, c)$, 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. There is a one-to-one correspondence between sequences in $\mathcal{S}$ and directed r-t paths in $\mathcal{D}$.

There are heuristics and exact algorithms for solving dynamic programs. For some problems, an exact dynamic programming algorithm runs in pseudo-polynomial time (Martello and Toth 1990, Floyd 1962). For others, the worst-case run-time is exponential, so state-space search heuristics are used in practice to try to find good solutions (Hart et al. 1968). Domain-independent dynamic programming is a recent framework that includes modeling and exactly solving dynamic programs (Kuroiwa and Beck 2024).

2.1.1 State-Space Relaxations

A *state-space relaxation* of a dynamic program is another dynamic program that is created by mapping the states of the original dynamic program into a more compact state space. A state-space relaxation can have more solutions than the original dynamic program and/or lower costs for existing solutions. There is a trade-off between the size of the state space and the how far the optimal solution value of the relaxation is from the optimal solution value of the original dynamic program. State-space relaxations were originally used to produce lower bounds for the well-known Traveling Salesman Problem (Christofides et al. 1981b).

Formally, a state-space relaxation of a dynamic program $P = (S_1, h_1, c_1)$ is another dynamic program (S_2, h_2, c_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 $s_1 \in S_2$, $c_2(s_1, u) = \min_{\{s_3 \in S_1 | \mu(s_3) = u, \mu(h_1(s_3,u)) = h_2(s_1,u)\}}\{c_1(s_3, u)\}$. It is difficult to find the state-space relaxation with a given size that produces the strongest possible bound, although a procedure to do so is called state-space ascent (Christofides et al. 1981b).

Algorithms for solving dynamic programs can use state-space relaxations. A branch-and-bound method uses state-space relaxations to produce lower bounds at each node (Christofides et al. 1981a). A sequential aggregation disaggregation algorithm uses state-space relaxations to find feasible solutions with a bounded optimality gap (Bean et al. 1987). A successive sublimation dynamic programming approach solves progressively refined state-space relaxations until an exact optimal solution is computed (Ibaraki and Nakamura 1994). A decremental state-space relaxation algorithm (Righini and Salani 2008), also known as a state-space augmenting algorithm (Boland et al. 2006), iteratively solves state-space relaxations that are strengthened in a way that removes the current solution from the relaxation.

2.2 Integer Linear Programming

An *integer linear program* is an optimization model that has decision variables, some of which are restricted to be assigned integer values, linear constraints, and a linear objective function. A *linear program* is a model with the same form, but without restrictions on the integrality of the decision variables. Consider an integer linear program of the following form, where $I = \{1, ..., m\}$ is a set of row indices, $J = \{1, ..., n\}$ is a set of column indices, $A \in \mathbb{Z}^{m,n}$ is a matrix, $c \in \mathbb{Z}^n$ is a vector of objective coefficients, and $b \in \mathbb{Z}^m$ is a vector of limits for the constraints.

$$\begin{aligned} \min_{x} \quad & \sum_{j \in J} c_j x_j & & \\ \text{s.t.} \quad & \sum_{j \in J} A_{ij} x_j \geq b_i & \forall i \in I & \\ & x_j \geq 0 & \forall j \in J & \\ & x_j \in \mathbb{Z} & \forall j \in J & \end{aligned} \tag{2.3}$$

The *linear program relaxation* of an integer linear program is the linear program created by removing the integrality constraints. Many algorithms exist to solve linear programs, including the simplex method (Dantzig et al. 1955) and interior point methods (Karmarkar et al. 1991). The linear program relaxation of the integer linear program above has the following form.

$$\begin{aligned} \min \quad & \sum_{j \in J} c_j x_j & & \\ \text{s.t.} \quad & \sum_{j \in J} A_{ij} x_j \geq b_i & \forall i \in I & \\ & x_j \geq 0 & \forall j \in J & \end{aligned} \tag{2.4}$$

The *dual linear program* of a (primal) linear program is a linear program that is derived from the primal linear program in a prescribed way. The dual linear program has a variable for each constraint in the primal linear program and a constraint for each variable in the primal linear program. The classical result of strong duality states that if the primal linear program has an optimal solution, then the dual linear program has an optimal solution and the optimal solution values are equal (Balinski and Tucker 1969). The dual linear program of the linear program above is as follows.

$$\begin{aligned} \max_{u} \quad & \sum_{i \in I} u_i b_i & & \\ \text{s.t.} \quad & \sum_{i \in I} A_{ij} u_j \leq c_j & \forall j \in J & \\ & u_i \geq 0 & \forall i \in I & \end{aligned} \tag{2.5}$$

A *cutting plane* is a linear inequality that can be added to a linear program relaxation to strengthen the formulation by removing fractional solutions, but without removing any solutions to the integer linear program. There are general cutting planes that can be applied to a generic linear program, such as Chvátal-Gomory inequalities (Chvátal 1973, Gomory 1960), and there are other problem-specific cutting planes such as rounded capacity cuts in vehicle routing problems (Augerat et al. 1998). The *separation problem* is to find a cutting plane that can be added to a linear program relaxation to remove a given fractional solution. A *cutting plane algorithm* iterates between solving

the linear program relaxation and the separation problem, and the algorithm terminates when an integer solution is found. A *finite cutting plane algorithm* solves an integer linear program in a finite number of iterations, which is possible with Chvátal-Gomory inequalities (Gomory 1960).

Branch-and-bound is an exact algorithm to solve integer linear programs by partitioning the set of feasible solutions (Beale 1979). The partition is often based on a single variable. For example, for $j \in J$ and $\pi \in \mathbb{Z}$, one subproblem includes the constraint $x_j \geq \pi$ and a second subproblem includes the constraint $x_j \leq \pi - 1$. Each subproblem is an integer linear program, so the process can be done recursively. Also, lower bounds on the optimal solution value of each subproblem are obtained by solving the linear programming relaxation, which can be used to reason that the optimal solution is not part of a subproblem, leading to a more efficient search.

Branch-and-cut uses cutting planes to improve the lower bounds produced for each subproblem during branch-and-bound (Padberg and Rinaldi 1987). A computational study shows that Chvátal-Gomory cuts can improve a basic branch-and-bound approach (Balas et al. 1996). State-of-the-art integer linear programming solvers use this approach (Jünger et al. 2009).

Variable fixing is the removal of a variable from an integer linear program by proving that the variable must be assigned to a particular value in an optimal solution. This reduces the number of variables in the integer linear program and can make it more easily solvable. A common variable fixing procedure relies on an optimal solution to the dual of the linear program relaxation and an upper bound on the optimal solution value (Nemhauser and Wolsey 1988). Any feasible solution to the dual of the linear program relaxation can also be used for variable fixing (Mitchell 1997). Consider the following example of variable fixing for the integer linear program above. Let u' be a feasible solution to the dual linear program of the linear program relaxation, and let χ be an upper bound on the optimal solution value. A variable x_j can be fixed to 0 if the following condition holds, as proven in Mitchell (1997).

$$\sum_{i \in I} u'_i b_i + c_j - \sum_{i \in I} u'_i A_{ij} > \chi \tag{2.6}$$

2.2.1 Column Generation

Column generation, also known as variable generation, is an algorithm for solving large-scale linear programs. It was originally used to solve a multicommodity flow problem (Ford and Fulkerson 1958) and then the cutting stock problem (Gilmore and Gomory 1961). One reason to use column generation is that it can solve linear programs without explicitly enumerating all the variables, making it possible to solve linear programs that would otherwise be intractable. Another reason is that, for integer linear programs, column generation can solve a Dantzig-Wolfe reformulation of a linear program, based on the Minkowski-Weyl decomposition of a convex polyhedron (Dantzig and Wolfe 1961, Weyl 1950). This reformulation can have an optimal solution value that is better than the linear programming relaxation (Geoffrion 1974).

Column generation solves a linear program by decomposing it into a restricted master problem and a pricing problem. The restricted master problem is the linear program restricted to a subset of variables, and the pricing problem is to find a variable that can improve the optimal solution value of the restricted master problem or to prove that such a variable does not exist. The algorithm terminates when the pricing problem returns that there is no improving variable. The hope is that this happens after a small number of iterations.

Consider the linear program from above. The *restricted master problem* has the following form, where $J'' \subseteq J$ is a subset of the variables.

$$\begin{aligned} \min_{x} \quad & \sum_{j \in J''} c_j x_j & & \\ \text{s.t.} \quad & \sum_{j \in J''} A_{ij} x_j \circ b_i & \forall i \in I & \\ & x_j \geq 0 & \forall j \in J'' & \end{aligned} \tag{2.7}$$

The *pricing problem* is an optimization problem that looks for a variable that can improve the optimal solution value of the restricted master problem. Technically, it checks if the optimal solution to the dual linear program of the restricted master problem is also feasible to the dual linear program of the original linear program. So, the problem can also be described as finding a constraint of the dual linear program that is violated by the optimal solution of the dual linear program of the restricted master problem, if one exists. This is equivalent to checking if the optimal solution value of the following problem is negative, where $u' \in \mathbb{R}^m$ is an optimal solution to the dual linear program of the restricted master problem.

$$\min_{j \in J} \quad c_j - \sum_{i \in I} u'_i A_{ij} \tag{2.8}$$

The pricing problem can be a challenging optimization problem, as it must consider all variables in the original linear program. However, column generation does not require the pricing problem to find a variable with the minimum reduced cost at each iteration. A heuristic can quickly find an improving variable for many iterations (Dumas et al. 1991). Still, an exact method for solving the pricing problem is needed to prove that an improving variable does not exist which allows column generation to terminate.

In some cases, the pricing problem is difficult to solve efficiently. In such cases, a column generation algorithm can instead solve a relaxation of the pricing problem (Fukasawa et al. 2006). Then, column generation needs to handle solutions to the relaxed pricing problem that are infeasible to the original pricing problem. It can add these 'infeasible' variables to the restricted master problem, which means that column generation may produce an infeasible solution, but it still gives a lower bound on the optimal solution value which can be useful for variable fixing or as part of a dual ascent method (Baldacci et al. 2011a). Otherwise, a method must be incorporated into the algorithm to strengthen the relaxation of the pricing problem and resolve it until the optimal solution is feasible to the original pricing problem (Desaulniers et al. 2008, Righini and Salani 2008).

Branch-and-price solves an integer linear program by using column generation to solve linear program relaxations at each node during branch-and-bound (Appelgren 1971). *Branch-and-cut-and-price* incorporates cutting planes into branch-and-price (Nemhauser and Park 1991). Cutting planes have been classified into two categories. A *robust cutting plane* can be added to the formulation without changing the structure or dimension of the pricing problem, while a *non-robust cutting plane* is likely to slow the algorithms to solve the pricing problem (de Aragao and Uchoa 2003). For non-robust cutting planes, it can be more effective to add weakened versions (Pecin et al. 2017a). Branch-and-cut-and-price has been particularly effective for scheduling (Bulhoes et al. 2020) and routing problems (Pessoa et al. 2020).

2.2.2 Lagrangian Relaxation

A *Lagrangian relaxation* of an integer linear program is a model that relaxes one or more constraints and penalizes their violation in the objective (Geoffrion 1974). The model is an optimization problem to find the vector of penalty values that gives the best objective value for the remaining optimization problem, called the Lagrangian subproblem. Relaxing the constraints should make this subproblem much easier to solve. Consider relaxing a subset of the indices $I' \subseteq I$ to an integer linear program. For a given vector of penalty values $\lambda \in \mathbb{R}^{|I'|}$, define the *Lagrangian subproblem* $L(\lambda)$ as follows.

$$\begin{aligned} \min_{x} \quad & \sum_{j \in J} c_j x_j + \sum_{i \in I'} \lambda_i (b_i - \sum_{j \in J} A_{ij} x_j) & \\ \text{s.t.} \quad & \sum_{j \in J} A_{ij} x_j \geq b_i & \forall i \in I \backslash I' \\ & x_j \geq 0 & \forall j \in J \\ & x_j \in \mathbb{Z} & \forall j \in J \end{aligned} \tag{2.9}$$

So, the Lagrangian relaxation has the following form.

$$\max_{\lambda \geq 0} \quad L(\lambda) \tag{2.10}$$

The Lagrangian relaxation has nice properties that makes it useful for solving an integer linear program. First, its optimal solution value is at least as good as the optimal solution value of the linear program relaxation, so it can be used as an alternative to the linear program relaxation in a method like branch-and-bound. Second, the optimal solution value changes nicely as the vector of penalty values changes; more technically, on the domain over which it is finite, the Lagrangian relaxation is piecewise linear, continuous, concave, and subdifferentiable. This enables some well-known algorithms to solve the Lagrangian relaxation. Using these properties, the Lagrangian relaxation has been successfully applied to many problems (Held and Karp 1970, Fisher 1973).

A *Lagrangian method* is an algorithm to solve the Lagrangian relaxation. We will focus on one such algorithm called subgradient descent. *Subgradient descent* begins with an initial feasible vector of penalties, and iteratively takes steps in the direction of a subgradient. At iteration k, let $\lambda^k \in \mathbb{R}^m$ be the vector of penalty values and x^k be an optimal solution to $L(\lambda^k)$. Subgradient descent updates the vector of penalty values according to the following equation, where $s^k \in \mathbb{R}$ is a step size to take in the direction of the given subgradient.

$$\lambda_i^{k+1} = \lambda_i^k + s^k (b_i - \textstyle\sum_{j \in J} A_{ij} x_j^k) \tag{2.11}$$

The choice of the initial solution and the step size at each iteration can greatly impact the computational performance of subgradient descent. A simple initial solution sets all penalties equal to 0, but another initial solution that uses information about the problem may improve the performance of the algorithm (Fisher 1981). A commonly used step size is the Polyak step size (Polyak 1969). Subgradient descent converges to an optimal solution when this step size is used (Polyak 1978). Another commonly used step size that makes subgradient descent provably converge is the Polyak target-value step size (Polyak 1969). This step size uses an estimate of the optimal solution value, denoted f^*.

$$s^k = \frac{(f^* - L(\lambda^k))}{||\lambda^k||^2} \tag{2.12}$$

Relax-and-cut incorporates cutting planes into an algorithm to solve the Lagrangian relaxation (Escudero et al. 1994). Cutting planes can improve the relaxation, but they are difficult to implement for two reasons (Lucena 2006). First, identifying a cutting plane requires an optimal solution, but a Lagrangian method like subgradient descent is an iterative approach that only converges to the optimal solution, and in practice may terminate before optimality. Second, after identifying a cutting plane, it likely needs to be relaxed and penalized in the objective to avoid making the Lagrangian subproblem difficult, which can change the optimal values of the other penalties. There are two main relax-and-cut approaches. *Delayed relax-and-cut* solves the Lagrangian relaxation with subgradient descent until some stopping criterion, then identifies cutting planes, decides to keep them in the formulation or likely penalizes their violation in the objective, and then restarts subgradient descent on the updated Lagrangian relaxation. *Non-delayed relax-and-cut* adds the cutting planes during subgradient descent without restarting the algorithm. These versions can improve the performance of a Lagrangian method, but the benefits can be limited due to the difficulties (Lucena 2005).

2.2.3 Arc Flow Formulations

An *arc flow formulation* is an integer linear program defined over a network (Ahuja et al. 1993). In particular, an arc flow formulation can be defined over the state-transition graph of a dynamic program (de Lima et al. 2022). To do this, additional resource costs must be included for the dynamic program transitions to represent coefficients for linear constraints. Let $G = \{g_j\}_{j=1}^{|J|}$ be the set of these additional resources where $g_j : \mathcal{A} \to \mathbb{R}$ for each $j \in J$. The model has a decision variable y_a for each arc $a \in \mathcal{A}$ that corresponds to the flow through the arc. The following is a general form for an arc flow formulation.

$$F: \quad \min \sum_{a \in \mathcal{A}} c(a) y_a \tag{2.13}$$

$$\text{s.t.} \sum_{a \in \mathcal{A}} g_j(a) y_a \geq b_j \qquad \forall\, j \in J \tag{2.14}$$

$$\sum_{a \in \delta^+(s)} y_a - \sum_{a \in \delta^-(s)} y_a = 0 \qquad \forall\, s \in \mathcal{N} \backslash \{r, t\} \tag{2.15}$$

$$y_a \in \mathbb{Z}_+ \qquad \forall\, a \in \mathcal{A} \tag{2.16}$$

An arc flow formulation can be solved directly with an integer linear programming solver or with branch-and-price (Valério de Carvalho 1999, Pessoa et al. 2010, Gouveia et al. 2019). Given an acyclic state-transition graph, a solution to the arc flow formulation can be decomposed into a set of paths and corresponding flow values (Ahuja et al. 1993). This transformation reveals that an arc flow formulation has an equivalent path-based formulation, which is an integer linear program with one variable for each path with appropriate objective and constraints. The path-based formulation is commonly solved with branch-and-price, so an arc flow formulation gives a direct way to solve an equivalent formulation without a decomposition. However, the state-transition graph can be large, preventing the arc flow formulation from being solved efficiently by a generic integer programming solver.

Iterative refinement algorithms solve large arc flow formulations by starting with an arc flow formulation over a state-space relaxation and using the optimal solution to update the relaxation. The main distinguishing features of such algorithms are the initial relaxation and the algorithm to

update the relaxation. The following are examples of such algorithms. An *iterative discretization algorithm* solves a vehicle routing problem by starting with an initial relaxation that rounds fractional values of distances and refines the relaxation by disaggregating some nodes (Macedo et al. 2011). A method called *dynamic discretization discovery* starts with an initial state-space relaxation that discretizes time in a way that maintains certain properties of its solutions and refines the relaxation by a two-step process that adds new time points and lengthens arcs (Boland et al. 2017). An *iterative aggregation and disaggregation* algorithm starts with an initial relaxation that aggregates nodes together and then the refinement step at least partially disaggregates these aggregations (Clautiaux et al. 2017).

2.3 Decision Diagrams

A *decision diagram* is a model for a discrete optimization problem that encodes a set of solutions and solution values as paths in a directed acyclic graph. Consider a similar problem to (2.1), where each x is a tuple (x_1,...,x_n) of discrete variables each with finite domains U_1,...,U_n respectively, and $f : \mathcal{S} \to \mathbb{R}$ is a cost function.

$$\min_{x \in \mathcal{S}} f(x) \tag{2.17}$$

Formally, a decision diagram is a directed acyclic graph $G = (\mathcal{N}, \mathcal{A})$ whose node set $\mathcal{N}$ is partitioned into layers $1, ..., n$ corresponding to the variables $x_1, ..., x_n$, including a root node in layer 1 and two terminal nodes in a final layer $n + 1$. For each node v in layer $i \in \{1, ..., n\}$, there is a directed arc in $\mathcal{A}$ for each value $u_i \in U_i$ which represents setting $x_i = u_i$. A *weighted decision diagram* also has costs for each arc $c : \mathcal{A} \to \mathbb{R}$. An *exact decision diagram* is a decision diagram such that the set of paths from the root node to terminal node is equivalent to the set of solutions $\mathcal{S}$, and the sum of the arc costs along the path for each solution $x \in \mathcal{S}$ equals $f(x)$ (Bergman et al. 2016b).

There are several forms of decision diagrams. A *binary decision diagram* has two arcs from each node and represents a Boolean function (Akers 1978, Lee 1959). A *reduced ordered binary decision diagram* is made more compact by merging nodes that are the roots of equivalent (isomorphic) subgraphs (Bryant 1986). An *edge-valued binary-decision diagram* represents an integer function or a psuedo-Boolean function (Lai et al. 1994). A *multi-valued decision diagram* allows more than two arcs to originate from each node (Srinivasan et al. 1990). Decision diagrams are closely related to dynamic programs, with one key difference being that the state-transition graph of a dynamic program is not reduced like decision diagrams (Hooker 2013).

An algorithm to solve an exact decision diagram requires compiling the graph and finding a shortest path from the root node to the terminal node, according to the arc costs. There are two common compilation methods. *Top-down compilation* performs a depth-first search over all possible assignments to the variables, and stores memory to reduce the decision diagram either during or after search (Andersen et al. 2007). *Incremental refinement* partitions some constraints that define the set of solutions $\mathcal{S}$, starts with building a decision diagram for some of these constraints, and then iteratively introduces the others (Huang and Darwiche 2005). Instead of creating a new decision diagram at each step of incremental refinement, it is possible to update the current decision diagram to incorporate new constraints by adding nodes and adding/removing arcs, referred to as *vertex splitting* (Hadzic et al. 2008).

The *variable ordering* of a decision diagram is the order of the discrete variables in the tuple that describes solutions. Different variable orderings can create decision diagrams of different sizes, so heuristics can be helpful to try to create a variable ordering that produces a small diagram to optimize over (Fujita et al. 1993). It is difficult to find the variable ordering that produces a decision diagram of smallest size, but algorithms exist to do this (Friedman and Supowit 1987) A *static* variable ordering is a single variable ordering that is used to compile a decision diagram and solve a problem. A *dynamic* variable ordering can be updated during the solving process (Rudell 1993).

2.3.1 Relaxed Decision Diagrams

A *relaxed decision diagram* is a decision diagram whose paths are equivalent to a superset of the feasible solution set $\mathcal{S}$ and/or the cost of each solution $x \in \mathcal{S}$ is at least as good as $f(x)$ (Andersen et al. 2007). By this definition, an optimal solution to a relaxed decision diagram gives a lower bound on the optimal solution value for the problem. Both top-down compilation (Bergman et al. 2016c) and iterative refinement (Hadzic et al. 2008) can be modified to produce a relaxed decision diagram. The top-down compilation approach merges nodes to maintain a limit on the maximum number of nodes on a layer. The iterative refinement approach can refine a subset of the constraints that define the feasible set of sequences.

An algorithm called *branch-and-bound with decision diagrams* solves an exact decision diagram by using relaxed decision diagrams for lower bounds at each node (Bergman et al. 2016c). *Peel-and-bound* is an extension of branch-and-bound with decision diagrams that improves computational performance by reusing compilation steps that are needed at more than one node of a branch-and-bound tree (Rudich et al. 2023). For these branch-and-bound methods, a *restricted decision diagram* that contains a subset of feasible solutions is used to obtain a feasible solution (Bergman et al. 2014b).

The origins of column elimination are in two works that each solve a discrete optimization problem with an arc flow formulation over a decision diagram. The initial work solves the vertex coloring problem (van Hoeve 2022). The algorithm begins with an arc flow formulation over a relaxed decision diagram that enumerates all independent sets in the graph and iterates between solving the arc flow formulation with an integer linear programming solver and refining the underlying relaxed decision diagram. The refinement algorithm removes an infeasible sequence from the relaxed decision diagram by making local changes to the network that introduces relatively few nodes and arcs. The algorithm is improved by solving the linear program relaxation of the arc flow formulation for many iterations before switching to solving the integer linear program. A second work solves a truck-drone routing problem (Tang and van Hoeve 2024). The arc flow formulation is a single path through an exact decision diagram with side constraints. The algorithm starts with common route relaxations from dynamic programming and iterates between solving the arc flow formulation and refining the route relaxation. One contribution is that the initial relaxed decision diagram is based on a state-space relaxation. Another contribution is to solve the linear program relaxation of the arc flow formulation via a Lagrangian relaxation, creating two new approaches. *LagAdapt* refines the relaxed decision diagram at each subgradient descent iteration. *LagRestart* collects infeasible sequences in a container and once a limit on the number of infeasible sequences is reached, these sequences are refined and subgradient descent is restarted.

Chapter 3

Column Elimination for the Capacitated Vehicle Routing Problem

This chapter introduces column elimination for solving the capacitated vehicle routing problem. It is based on the work by Karahalios and van Hoeve (2023b). This work advances column elimination from prior papers in a few significant ways. First, it furthers the Lagrangian method proposed in Tang (2021). In particular, the Lagrangian subproblem for the capacitated vehicle routing problem is a minimum cost flow problem while the Lagrangian subproblem in the previous work is a shortest path problem. This work implements an efficient successive shortest paths algorithm to solve this more challenging Lagrangian subproblem. Second, cutting planes are introduced to strengthen the dual bound. Third, variable fixing is incorporated to reduce the size of the problem.

3.1 Introduction

The capacitated vehicle routing problem (CVRP) can be stated as follows Toth and Vigo (2014). Given a set of locations each with a specified weight and a fleet of vehicles each with a specified capacity, the problem asks to design a route for each vehicle such that each location is visited by a vehicle, for each truck the total weight of its visited locations does not exceed the capacity, and the sum of the vehicle route lengths is minimized. It is a central problem in logistics and has become increasingly important over the last decade due to the rise of last-mile delivery applications. The CVRP is among the most studied NP-hard combinatorial optimization problems and finding provably optimal solutions remains a challenge in practice. Current state-of-the-art exact methods can solve up to around 200 locations optimally within a reasonable of time, with branch-cut-and-price (BCP) methods performing particularly well (Fukasawa et al. 2006, Baldacci et al. 2011b, Pecin et al. 2017b, Pessoa et al. 2018, 2020).

BCP is an effective method for solving generic large-scale integer programming models (Barnhart et al. 1998). It relies on column generation to solve the linear programming relaxation: working with a restricted set of variables (or columns), column generation iteratively adds new variables to the model until an optimal basis is found. Despite its successes, column generation has some

weaknesses. For example, it may take many iterations to converge to the optimal solution due to dual degeneracy of the intermediate solutions. Furthermore, branching decisions or cutting planes that strengthen the relaxation may complicate the pricing problem that finds new variables.

We study an alternative approach that does not rely on a pricing problem, thereby avoiding the potential drawbacks of column generation mentioned above. Instead of using a restricted set of columns, column elimination works with a *relaxed* set of columns, from which infeasible ones are iteratively eliminated. As the total number of columns can be exponentially large, we use state-space relaxations to compactly represent and manipulate the set of columns. This method was first introduced for the graph coloring problem in (van Hoeve 2020c, 2022), then applied to the traveling salesperson problem with a drone (Tang 2021, Tang and van Hoeve 2024), and later termed 'column elimination' (van Hoeve and Tang 2022).

The main focus of this work is to develop strong *dual bounds* for the CVRP using column elimination. As will be formalized later, column elimination and column generation will produce the same dual bound if they work from the same underlying state-space relaxation. Column elimination can potentially produce stronger bounds than the initial state-space relaxation as it can remove infeasible columns beyond those that are excluded by the initial state-space relaxation. Moreover, column elimination allows a more liberal use of cutting planes to strengthen the relaxation. We show how existing cuts from the column generation literature can be expressed directly into the column elimination model, while in addition the structure of the arc flow formulation permits us to develop new cuts. The novel contributions include developing an efficient solution method via a Lagrangian reformulation, introducing cuts to column elimination, incorporating variable fixing, and showing how column elimination can produce bounds competitive with state-of-the-art solvers for the CVRP.

The paper is organized as follows. In Section 3.2 we present the column formulation of the CVRP. Section 3.3 describes the dynamic program and arc flow formulation. The column elimination procedure is presented in Section 3.4. Section 3.5 presents our Lagrangian relaxation. In Section 3.6 we describe how cutting planes can be added to strengthen the model. Section 3.7 presents a reduced cost-based variable fixing procedure to reduce the size of the model. We conduct experimental results in Section 3.8 and conclude in Section 3.9.

3.2 Column Formulation for CVRP

We first give a formal definition of the CVRP (Toth and Vigo 2014). Let $U = \{0, 1, \dots, n\}$ be a set of locations with the depot as 0. Each location has demand $q_i \geq 0$ and $T_{ij} > 0$ is the distance from i to j. Each vehicle has capacity Q. A *route* is a sequence of locations $[u_1, u_2, \dots, u_k]$ 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., $\sum_{i=1}^{k-1} T_{u_i u_{i+1}}$. Let K be the number of (homogeneous) vehicles, each with capacity Q. 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.

The column formulation for the CVRP is based on the set R of all feasible elementary routes (Balinski and Quandt 1964). We let d_r denote the distance of route $r \in R$. We define a matrix $M^{n \times |R|}$ such that $M_{ir} = 1$ if location $i \in \{1, 2, \dots, n\}$ belongs to route $r \in R$, and $M_{ir} = 0$ otherwise. That is, each column vector in M corresponds to a route. Lastly, we define a binary decision variable x_r

for each $r \in R$. The column formulation of the CVRP is:

$$\begin{aligned} \min \quad & \sum_{r \in R} d_r x_r \\ \text{s.t.} \quad & \sum_{r \in R} M_{ir} x_r = 1 \qquad \forall i \in \{1, 2, \ldots, n\} \\ & \sum_{r \in R} x_r = K \\ & x_r \in \{0, 1\} \qquad \forall r \in R. \end{aligned} \tag{3.1}$$

This model is also known as the *set partitioning* formulation. In practice the set of routes R often has exponential size, which restricts the direct application of the set partitioning model to very small instances. Branch-and-price (Barnhart et al. 1998) provides a more scalable approach by using a column generation procedure to solve the continuous linear programming relaxation of (3.1).

Column generation starts by solving the linear programming relaxation of the set partitioning model defined on a (small) subset of variables, known as the *restricted master problem*. Using the dual variables of the optimal solution it then solves a *pricing problem* to find a new variable with a negative reduced cost. This process continues until no more improving variables exist and the restricted master has a provably optimal basis. To ensure integer feasibility, column generation is embedded into a systematic search.

Solving the pricing problem for the CVRP is not straightforward, because it corresponds to the NP-hard elementary shortest path problem with resource constraints (Irnich and Desaulniers 2005). It can be solved with a dynamic programming labeling algorithm, which is however limited by the exponential size of the state-space. A computationally efficient alternative is to relax the pricing problem to find a shortest path that is not necessarily elementary, i.e., certain locations can be visited more than once (Christofides et al. 1981a). Recent examples include the q-route relaxation (Fukasawa et al. 2006) and the ng-route relaxation (Baldacci et al. 2011b). The linear programming model from route relaxations can be further strengthened by adding cutting planes to the restricted master problem (Pecin et al. 2017b).

3.3 Arc Flow Formulation for CVRP

The key ingredient of the column elimination procedure is to compactly represent the set of routes R via the state-transition graph of a dynamic program. The CVRP can then be formulated as an arc flow formulation over the dynamic program as in prior works (van Hoeve 2022, Tang and van Hoeve 2024).

3.3.1 Dynamic Program for Storing Routes

We define $P_{ESPRC} = (S, h, c)$ as a dynamic program that encodes the set of routes. It is equivalent to the dynamic program for the elementary shortest path problem with resource constraints (Irnich and Desaulniers 2005). 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. The transition and cost functions are defined for two cases: visiting a location and returning to the depot. For each state $s = (\text{NG}, w, v)$ and

element $i \in U$ such that $i \notin \text{NG}$, $i > 0$, and $w + q_i \leq Q$, define the transition function as

$$h(s, i) = (\text{NG} \cup \{i\}, w + q_i, i)$$

and $c(s, i) = T_{vi}$. For each state $s \in S$, define $h(s, 0) = t$ and cost $c(s, 0) = T_{v0}$. Other combinations of states and decisions are infeasible. Define $g_j(s, i) = [\![i = j]\!]$ for each $j = 1, ..., m$.

3.3.2 State-Space Relaxations

The two most-used state-space relaxations for the CVRP in the column generation literature are the q-route relaxation (Fukasawa et al. 2006) and the ng-route relaxation (Baldacci et al. 2011b). Both are based on the dynamic program P, but relax the set of visited locations S.

The *q-route relaxation* maintains the last q visited locations. We define the dynamic program P_q for this state-space relaxation as follows. Define a state as (SQ, w) where w is defined as above, and $SQ = [u_1, \ldots, u_q]$ is a sequence of locations. The initial state is $([\text{-}, \ldots, \text{-}], 0)$. Given a state $s = (SQ, w)$ and label $i \in V$ such that $i \notin SQ$ and $w + q_i \leq Q$, we define the transition function as

$$h_q(s, i) = ([u_2, \ldots, u_q, i], w + q_i)$$

with associated transition cost function $c_q((SQ, w), i) = T_{i_q i}$.

For the *ng-route relaxation*, we assume that a set $N_i \subseteq U$ of size g exists for each $i \in \{1, \ldots, n\}$. The set N_i must include i and typically represents the g locations closest to i. We define the dynamic program P_{ng} for this state-space relaxation as follows. Define a state as (NG, w, e) where the 'no-good' set $NG \subseteq V$ is a subset of visited locations, and w and e are as above. The initial state is $(\varnothing, 0, 0)$. Given a state $s = (NG, w, e)$ and label $i \in V$ such that $i \notin NG$ and $w + q_i \leq Q$, we define the transition function as

$$h_{ng}((NG, w, e), i) = ((NG \cup \{i\}) \cap N_i, w + q_i, i)$$

with associated transition cost function $c_{ng}((NG, w, e), i) = T_{ei}$. Observe that P_q and P_{ng} forbid cycles of length at most q and g, respectively.

3.3.3 Properties of State-Space Relaxations

We show two useful properties of the above dynamic programs. Given a dynamic program P, let $\mathcal{S}_P$ be its set of solutions and $f_P : \mathcal{S}_P \to \mathbb{R}$ be its cost function. Recall that d_r represents the distance of route $r \in R$.

Proposition 1. $\mathcal{S}_{P_{ESPRC}} = R$ and for each $x \in \mathcal{S}_{P_{ESPRC}}$, $f_{ESPRC}(x) = d_x$.

Proof. P_{ESPRC} encodes elementary paths and represents all possible feasible routes and their associated distances. □

Proposition 2. $\mathcal{S}_P \subseteq \mathcal{S}_{P_q}$, $\mathcal{S}_P \subseteq \mathcal{S}_{P_{ng}}$ and for each $x \in \mathcal{S}_{P_{ESPRC}}$, $f_{ESPRC}(x) = f_q(x) = f_{ng}(x)$

Proof. Both P_q and P_{ng} encode a relaxation that contains elementary paths, and therefore represent a superset of all possible feasible routes. Because they both maintain the last visited location their cost functions are not relaxed. □

3.3.4 Arc Flow Formulation

We next reformulate the set partitioning model (3.1) as an arc flow formulation. Let $D = (\mathcal{N}, \mathcal{A})$ be a state-transition graph for a dynamic program that represents a set of routes for the CVRP. We introduce a 'flow' variable $y_a \geq 0$ for each $a \in \mathcal{A}$. We denote the set of arcs in $\mathcal{A}$ that correspond to a transition for location i by $\mathcal{A}^i$. The model is as follows:

$$F(D): \quad \min \sum_{a \in \mathcal{A}} c_a y_a \tag{3.2}$$

$$\text{s.t.} \sum_{a \in \delta^-(u)} y_a - \sum_{a \in \delta^+(u)} y_a = 0 \qquad \forall u \in \mathcal{N} \setminus \{r, t\} \tag{3.3}$$

$$\sum_{a \in \mathcal{A}^i} y_a = 1 \qquad \forall i \in V \setminus \{0\} \tag{3.4}$$

$$\sum_{a \in \delta^+(r)} y_a = K \tag{3.5}$$

$$y_a \in \{0, 1\} \qquad \forall a \in \mathcal{A}. \tag{3.6}$$

The objective function (3.2) minimizes the sum of all arc costs. The 'flow conservation' constraints (3.3) ensure that the solution is a collection of labeled r-t paths. Constraints (3.4) ensure that all locations are visited once. Constraint (3.5) enforces that exactly K units of flow originate from r. The binary constraints (3.6) complete the formulation.

Let D_P be the state-transition graph for a dynamic program P.

Theorem 1. $F(D_{P_{ESPRC}})$ is an exact formulation of the CVRP.

The proof relies on the fact that the dynamic programming model represents all possible routes, that each solution of the network flow model consists of exactly K r-t paths, and that each r-t path corresponds to a feasible route.

Corollary 1. $F(D_{P_q})$ and $F(D_{P_{ng}})$ yield a dual bound for the CVRP.

In the remainder of this paper, we will use the continuous linear programming relaxation of model $F(D)$, referred to as $LP(F(D))$, which is obtained by replacing the integrality constraints (3.6) by $0 \leq y_a \leq 1$ for all $a \in \mathcal{A}$.

3.4 Column Elimination Procedure

We present a schematic representation of column elimination in Figure 3.1. Starting with an initial state-transition graph of a state-space relaxation, the column elimination procedure iteratively 1) solves the constrained network flow model $F(D)$, 2) decomposes the solution into paths (routes), 3) identifies infeasible paths and removes them from D, and repeats. The process terminates when no infeasible paths are detected in which case $F(D)$ is solved to optimality. It can also terminate earlier when the dual bound matches a given (or heuristically generated) primal bound, or when a different stopping criterion such as a time or memory limit is met. The procedure can utilize either the integer model $F(D)$ or its continuous relaxation $LP(F(D))$; using $LP(F(D))$ would solve the continuous linear programming relaxation of (3.1), but could be embedded in branch-and-bound to solve the full problem.

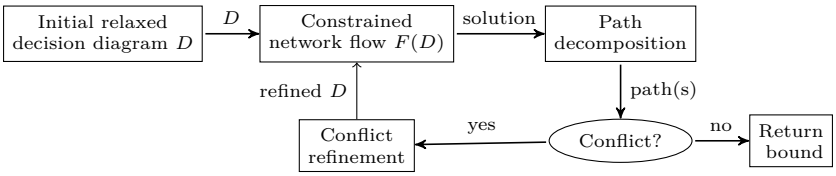


Figure 3.1: Overview of the column elimination framework, adapted from Tang and van Hoeve (2024).

Any (existing) state-space relaxation for the CVRP can be applied to construct the initial state-transition graph. Recall that model P_{ESPRC} has state definition (S, w, e), and each of these three elements can potentially be relaxed to define a state-space relaxation. The q-route and ng-route relaxations only relax the elementarity constraint, i.e., the set S. This means that conflicts will only come in the form of repeated labels; each path respects the truck capacity constraint and the route costs are exact. For a state-transition graph D derived from such a state-space relaxation, $F(D)$ is an exact formulation for the CVRP. In practice, we prefer using a relaxation that is relatively small and provides a 'good' starting point in terms of bound quality from $LP(F(D))$. In our experiments, we therefore use P_q with $q = 1$ and P_{ng} with $g = 2$ to initialize the state-transition graph, with the latter performing best.

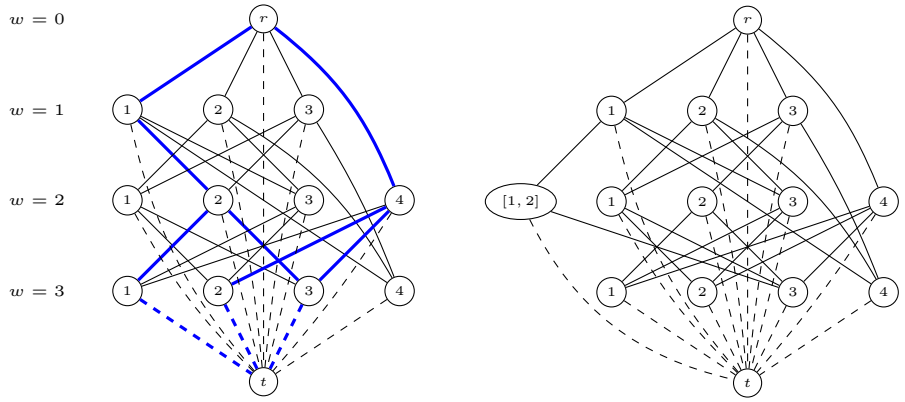
Given the initial state-transition graph D, we solve the associated model $LP(F(D))$, apply a path decomposition of the solution, and inspect the paths for any conflicts. For our choice of state-space relaxations, the only conflicts arise from repetition of locations along a path. To remove a conflict, we follow the (partial) path elimination process outlined in (van Hoeve 2022): it essentially separates the path by introducing a new node at each layer, and removing the arc associated with the repeated label. During this process, we will update the state information of the nodes along the separated path. We illustrate conflict separation in the next example, and refer to (van Hoeve 2022) for more details.

Locations $V = \{0, 1, 2, 3, 4\}$
Depot $= 0$
Demands $q_1 = q_2 = q_3 = 1$, $q_4 = 2$
Number of trucks $K = 2$
Vehicle capacity $Q = 3$

l_{ij}	0	1	2	3	4
0	0	5	10	5	10
1	5	0	10	10	15
2	10	10	0	10	15
3	5	10	10	0	10
4	10	15	15	10	0

Figure 3.2: Input data for the CVRP instance in Example 3.4.1.

Example 3.4.1. Consider the CVRP instance with the problem data given in Figure 3.2. The integer optimal solution uses routes $[0, 1, 2, 0]$ and $[0, 3, 4, 0]$ with total distance 50. The state-transition graph for P_q with $q = 1$ is presented in Figure 3.3(a). Each node in the diagram is associated with its SQ state, i.e., the last visited location. The weights are omitted from the states; instead nodes with the same cumulative weight are represented in the same layer. For clarity, we also omit the arc labels and arc costs. Arcs into t correspond to terminating a route and are dashed. The optimal solution to the linear programming relaxation of $F(D)$ yields dual bound 48.333 and uses the following arc-label specified paths: path $(1, 2, 1, 0)$ with flow value $\frac{1}{3}$, path $(1, 2, 3, 0)$ with flow value $\frac{1}{3}$, path $(4, 2, 0)$ with flow value $\frac{1}{3}$, and path $(4, 3, 0)$ with flow value $\frac{2}{3}$.



a. Relaxed decision diagram from DP_{SQ_1}.

b. Refined decision diagram.

Figure 3.3: State-transition graphs for the CVRP instance in Example 3.4.1. Figure (a) depicts the state-transition graph for P_q with $q = 1$. The optimal solution to model $LP(F(D))$ is indicated by thick blue arcs. Figure (b) represents the refined decision diagram after eliminating the partial path $[1, 2, 1]$ that contains a conflict.

The first path contains a conflict: label 1 is repeated. We separate this conflict by rerouting the path to a new node with state $SQ = [1, 2]$ remembering location 1 in addition to 2. As a result, we eliminate the arc with label 1 from the new state. The refined state-transition graph is depicted in Figure 3.3(b). It yields a dual bound of value 50, which is optimal.

3.5 Lagrangian Relaxation

Because the state-transition graph can grow large in size, solving the arc flow formulation can become the computational bottleneck of our method, even when we consider the linear programming relaxation. To potentially solve the model more efficiently, we consider solving a Lagrangian relaxation, similar to (Tang 2021, Tang and van Hoeve 2024), that has optimal bound equivalent to $LP(F(D))$. We obtain our Lagrangian relaxation of the constrained network flow model by dualizing constraints (3.4) that require each location to be visited once. We introduce a Lagrangian multiplier λ_i for each $i \in U \backslash \{0\}$, and define the Lagrangian relaxation as

$$L(D, \lambda): \quad \min \sum_{a \in \mathcal{A}} c_a y_a + \sum_{i \in V \setminus \{0\}} \lambda_i (1 - \sum_{a \in \mathcal{A}^i} y_a) \tag{3.7}$$

$$\text{s.t.} \sum_{a \in \delta^-(u)} y_a - \sum_{a \in \delta^+(u)} y_a = 0 \qquad \forall u \in \mathcal{N} \setminus \{r, t\} \tag{3.8}$$

$$\sum_{a \in \delta^+(r)} y_a = K \tag{3.9}$$

$$y_a \in \{0, 1\} \qquad \forall a \in \mathcal{A}. \tag{3.10}$$

The objective function (3.7) can be rewritten as

$$\min \sum_{a \in \mathcal{A}} c_a y_a - \sum_{i \in V \setminus \{0\}} \lambda_i \sum_{a \in \mathcal{A}^i} y_a + \sum_{i \in V \setminus \{0\}} \lambda_i =$$
$$\min \sum_{a \in \mathcal{A}} (c_a - \lambda_{\ell_a}) y_a + \sum_{i \in V \setminus \{0\}} \lambda_i.$$

As a consequence, for fixed λ, the Lagrangian relaxation can be solved as a (continuous) minimum-cost network flow problem over the state-transition graph, using $c_a - \lambda_{\ell_a}$ as the cost for arc $a \in \mathcal{A}$, yielding an integer optimal solution. In fact, given constraints (3.9) and the unit capacity constraints on the arcs, each solution consists of K arc-disjoint r-t paths. By applying the successive shortest paths (SSP) algorithm (Ahuja et al. 1993) to solve $L(D, \lambda)$ we obtain the following result:

Lemma 1. Given a state-transition graph $D = (\mathcal{N}, \mathcal{A})$ and fixed λ, the Lagrangian relaxation $L(D, \lambda)$ can be solved in $O(K(|\mathcal{N}| \log(|\mathcal{N}|) + |\mathcal{A}|))$ time.

We also implemented a dedicated algorithm, based on the 'minimum update Successive Shortest Paths' (muSSP) algorithm that was developed for specific directed acyclic graphs in the content of multi-object tracking in computer vision (Wang et al. 2019). Although graphs with a slightly different structure are considered in (Wang et al. 2019), the algorithm generalizes to our case: weighted directed acyclic graphs with one source (the root), one sink (the terminal), and unit capacities. The muSSP algorithm leverages the fact that most updates to the shortest path tree through Dijkstra's algorithm are not useful, and it aims instead to make minimal updates to the shortest path tree. While it has the same theoretical worst-case time complexity as the SSP, in practice the muSSP algorithm can be an order of magnitude more efficient than the standard SSP algorithm.

The Lagrangian subproblem $\max_\lambda L(D, \lambda)$ finds the multipliers that provide the best Lagrangian bound. Because the objective in $L(D, \lambda)$ is concave and piecewise linear, the dual can be solved via a subgradient method. At each iteration k of the subgradient method, one choice for a subgradient that we will use is γ^k such that $\gamma_i^k = (1 - \sum_{a \in \mathcal{A}^i} y_a^k)$ where y_a^k is the solution to $L(D, \lambda^k)$. Then we update the dual multipliers for the next iteration as $\lambda^{k+1} = \lambda^k + \alpha^k \gamma^k$, where we use an estimated Polyak step size α^k (Boyd et al. 2003). Note that the initial choice of multipliers λ^0 can be important for solving the dual quickly (Bertsekas 2014).

We remark that the optimal Lagrangian dual bound is equal to the optimal linear programming bound from $LP(F(D))$, when both apply the same state-transition graph. Moreover, when the column elimination process uses $LP(F(D))$ or $L(D, \lambda)$, its bound at termination is equal to the column generation bound of the set partitioning model (3.1), assuming that all methods use the same underlying dynamic programming formulation, as observed in Tang and van Hoeve (2024). That is, the state-transition graph is for the the same dynamic programming formulation in its construction as column generation uses in the pricing problem.

Lastly, we note that in each iteration of the subgradient method for solving the Lagrangian dual the solution can potentially be used to identify and separate conflicts. Similar to (Tang 2021), we separate these conflicts in batches of size 100, after which we restart the Lagrangian process.

3.6 Cutting Planes

Results from the literature show that the LP relaxation of the set partitioning formulation for CVRP, solved via column generation, frequently has a 1-4% optimality gap. To further strengthen the LP

relaxation several classes of valid inequalities can be added. According to the literature, the most effective are rounded capacity cuts, strengthened comb inequalities, and subset-row cuts (Lysgaard 2003, Fukasawa et al. 2006, Pecin et al. 2017b). The first two types of cuts are called *robust* in the column generation literature because they do not affect the runtime of the pricing problem, while the subset-row cuts are not robust. We next show how rounded capacity cuts and strengthened comb inequalities can be implemented in our arc flow formulation $LP(F(D))$, as well as a generalization of subset-row cuts as a type of clique cut.

Rounded capacity cuts ensure that a subset of locations S is visited by a sufficient number of trucks to meet its aggregate demand. In column generation these cuts can be added to model (3.1) so long as the underlying routes are stored for each $r \in R$. Let p_r^S be the number of times route r uses an edge between S and $V \backslash S$, and let $k(S) = \lceil \frac{1}{Q} \sum_{i \in S} q_i \rceil$. The cut added to the restricted master problem is $\sum_{r \in R} p_r^S x_r \geq 2k(S)$, and the associated dual variable is added to controls in the dynamic program for the pricing problem that correspond to a route traversing an edge between S and $V \backslash S$. To add this cut in column elimination, let A^S be the set of arcs $a \in A$ such that $\ell(a) \in S$ and the node u that is the head of a has state with last visited location $i \in V \backslash S$, or the other way around with $\ell(a) \in V \backslash S$ and $i \in S$. A rounded capacity cut for set S can be modeled by adding to $LP(F(D))$ the following inequality: $\sum_{a \in A^S} y_a \geq 2k(S)$. Note that when solving $LP(F(D))$ using the Lagrangian formulation, this constraint can be dualized.

Strengthened comb inequalities are a generalization of comb inequalities that have been proven highly useful for solving the Traveling Salesman Problem (Lysgaard et al. 2004). A strengthened comb inequality is defined by a handle set of locations H and teeth sets of locations T_t for $t \in \{1, ..., T\}$. Let $S(H, T_1, ..., T_T)$ be the appropriately defined right hand side for the inequality (Lysgaard et al. 2004). In column generation, this cut also requires storing the underlying routes and can be added to the restricted master problem as $\sum_{r \in R} p_r^H x_r + \sum_{t \in T} \sum_{r \in R} p_r^{T_t} x_r \geq S(H, T_1, ..., T_T)$. The associated dual variable is then added to controls in the dynamic program for the pricing problem that correspond to traversing edges with one endpoint in H or one of T_i and the other endpoint not in that set. In column elimination, a strengthened comb inequality with handle H and teeth T_t can be modeled by adding to $LP(F(D))$ the following inequality: $\sum_{a \in A^H} y_a + \sum_{t \in \{1,...,T\}} \sum_{a \in A^{T_t}} y_a \geq S(H, T_1, ..., T_T)$. This constraint can also be dualized when using the Lagrangian formulation to solve $LP(F(D))$.

Subset row cuts are non-robust cuts that have been successfully applied to the CVRP. In particular, the limited memory subset row cuts are an important part of the success of the column generation method in (Pecin et al. 2017b). Since the arc flow formulation representation does not have a matrix view of the set of routes, subset row cuts do not directly translate to the $LP(F(D))$ model. However, they can be generalized by a class of clique cuts on a specific conflict graph (Balas and Ho 1980). These cuts are non-robust and have been used in (Baldacci et al. 2008) but not until the problem size has been reduced. The structure of our state-transition graph allows column elimination to implement a specific version of these cuts. Let $D = (\mathcal{N}, \mathcal{A})$ be a state-transition graph as defined above. The conflict graph $G^C = (\mathcal{N}, A^C)$ is defined on node set $\mathcal{N}$. Its arc set A^C contains all arcs (i, j) such that 1) the set of visited locations in the states associated to nodes i and j have a non-empty intersection, and 2) nodes i and j never appear on the same directed path in D. A *clique cut* states that the flow through nodes in a clique of G^C must be at most 1:

Theorem 2. Let $\mathcal{C}$ be a clique in the conflict graph G^C derived from a state-transition graph D. The associated *clique cut* $\sum_{i \in \mathcal{C}} \sum_{a \in \delta^-(i)} y_a \leq 1$ is a valid inequality for model $LP(F(D))$.

Proof. By construction of G^C, each pair of nodes $i, j \in \mathcal{C}$ has at least one common visited location

(say u) in their associate states, and there is no directed path between i to j in D. Suppose that for an integral optimal solution we have $\sum_{a\in\delta^-(i)\cup\delta^-(j)} y_a > 1$. This means that location u is visited twice, which cannot occur in an optimal solution: a contradiction. □

Given G^C and a set of cliques in G^C, clique cuts can be easily separated for $LP(F(D))$ by evaluating whether a given fractional solution violates a cut. Because a solution to the Lagrangian model $L(D,\lambda)$ is integral, we cannot directly use it to separate any cuts. In (Anstreicher and Wolsey 2009) it is shown that a weighted average of the subproblem solutions converges to an optimal primal solution and we apply this method to identify valid inequalities.

3.7 Reduced Cost-Based Arc Fixing

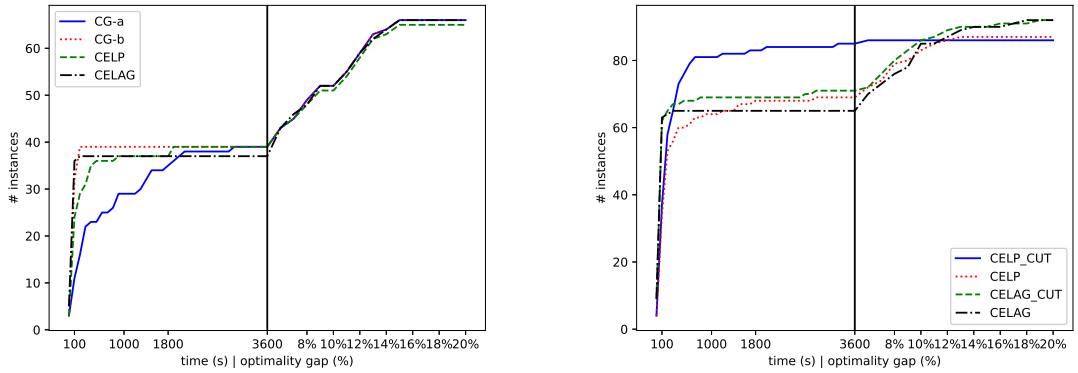
Variable fixing based on reduced costs is often applied to reduce the problem size of integer programs (Nemhauser and Wolsey 1988), including the CVRP (Irnich et al. 2010, Pecin et al. 2017b). It uses a feasible dual solution and suitably small optimality gap to set the value of a primal variable equal to 0 (Ahuja et al. 1993, Fischetti and Toth 1989, Lodi et al. 2003). We develop an arc fixing method for the $LP(F(D))$ model, using similar arguments as (Pecin et al. 2017b).

Let $D = (\mathcal{N}, \mathcal{A})$ be a state-transition graph that contains a set of routes $R' \subseteq R$. Consider a feasible dual solution (ν, κ) to the LP relaxation of the set partitioning model (3.1) over R', where ν correspond to the 'set partitioning' constraints and κ to the 'number of trucks' constraint. For each arc $a \in \mathcal{A}$ we define a 'reduced cost distance' $rc(a) = l_a - \nu_{\ell_a}$. For each node $u \in \mathcal{N}$, we define $sp^{\downarrow}_u$ as the shortest r-u path in D with respect to the reduced cost distances, and similarly define $sp^{\uparrow}_u$ to be the shortest u-t path in D.

Theorem 3. Consider arc $a = (v_1, v_2) \in \mathcal{A}$. Let $v(\nu,\kappa)$ be the dual solution value, and let UB an upper bound on (3.1). If $v(\nu,\kappa) + sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(a) - \kappa > UB$, then arc a can be fixed to have flow 0 in $F(D)$ and accordingly in $LP(F(D))$.

Proof. Given (ν,κ), each route $\overline{r} \in R'$ in the LP relaxation of (3.1) has reduced cost $rc(\overline{r}) = d_{\overline{r}} - \sum_{i=1}^{n} M_{i\overline{r}}\nu_i - \kappa$. Each $\overline{r}$ corresponds to a path $p = \{a_1, ..., a_l\}$ in D, so $rc(\overline{r})$ can be decomposed into $rc(\overline{r}) = \sum_{i=1}^{l} rc(a_i) - \kappa$. For all p that contain arc a, let p' be the path that corresponds to the route r' with lowest reduced cost. Denote $rc(r') = sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(a) - \kappa$. Now for sake of contradiction assume an optimal solution to $F(D)$ has $y_a = 1$. Then some path p'' in D that contains arc a will have flow of 1, so we can consider this as some $x_{r''} = 1$ in an optimal solution to (3.1). To construct the remainder of an optimal solution to the LP relaxation of (3.1) we can solve this LP relaxation with constraints for locations in r'' removed and only requiring $K-1$ trucks. Because (ν,κ) remains feasible to the dual of this updated problem and has value $v(\nu,\kappa) - \sum_{i=1}^{n} M_{ir''}\nu_i - \kappa$, it gives a valid lower bound on (3.1) that contradicts UB, namely $v(\nu,\kappa) - \sum_{i=1}^{n} M_{ir''}\nu_i - \kappa + d_{r''} = v(\nu,\kappa) + rc(r'') \geq v(\nu,\kappa) + rc(r') \geq v(\nu,\kappa) + sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(a) - \kappa > UB$. □

Note that while Theorem 3 relies on the set partitioning model (3.1) to build the reduced cost argument, we can use the optimal dual solution to $LP(F(D))$ in the application of the theorem. When solving $LP(F(D))$ with a standard linear programming solver, we can use the feasible dual from the previous iteration – which remains feasible even with cuts and separations – to fix arcs. One important note is that these fixed arcs are reintroduced if separation happens before the next iteration, as the change in the state-transition graph structure may disrupt previous arc fixing arguments. When solving $LP(F(D))$ via its Lagrangian relaxation $L(D,\lambda)$, we must ensure that we



a. Baseline CE vs. CG

b. Adding Cuts to CE

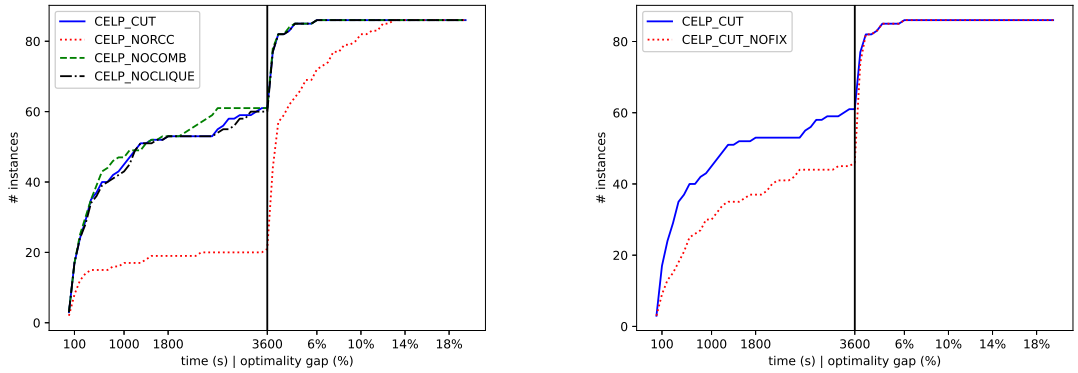
Figure 3.4: a) Comparing column generation over P_q for $q = 2$ with column elimination starting from P_q with $q = 1$ and using different methods to solve $LP(F(D))$ without any added cuts. b) Performance plot for adding cuts to CELP and CELAG.

work with a feasible dual solution. In addition, we include a dual variable for constraint (3.10) and set it to its maximum value while ensuring dual feasibility.

3.8 Experimental Results

We use the benchmark set of CVRP instances from `http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`, including the new challenge set of instances from (Uchoa et al. 2017). All experiments are run on an Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz. We use CPLEX version 22.1 (Manual 1987) as a linear programming solver and change 3.4 to $\geq$ to help find an initial feasible solution. We use the package CVRPSEP (Lysgaard 2003) to heuristically find rounded capacity cuts and strengthened comb inequalities when given a fractional primal solution. At each iteration we add at most 10 robust capacity cuts and 5 strengthened comb inequalities, using the most violated ones possible. We use Cliquer (Östergård 2002) to heuristically find large weighted cliques in the conflict graph used to derive clique cuts.

Comparing Column Elimination and Column Generation. We compare column generation over P_q with $q = 2$ with column elimination starting from the P_q with $q = 1$ state-space relaxation and eliminating cycles of size 2. Doing so, the final bounds are the same, which allows us to compare how quickly column generation and column elimination reach the optimal bound. We implement a vanilla version of column generation that starts with a small set of greedily chosen routes and solves the pricing problem as shortest paths through the pre-compiled state-transition graph for P_q with $q = 2$. We compare column generation not including and including time to compile the state-transition graph (CG-a, CG-b), column elimination using CPLEX (CELP), and column elimination using a subgradient method over the Lagrangian dual (CELAG). We run each method for 3,600 seconds over benchmark sets A, B, E, F, M, P. We remove instances when the state-transition graph for P_q with $q = 2$ does not finish compiling. Arc fixing uses the best known solution as an upper



a. Evaluating Individual Cuts

b. Evaluating Arc Fixing

Figure 3.5: a) Evaluating the performance of individual cuts on CELP_CUT. b) Performance plot of CELP_CUT with and without arc fixing

bound and is used in CELP but not in CELAG. Lower bounds for column generation are computed before termination as in (Wolsey 2020). Figure 3.4.a is a performance plot of the number of instances solved to within a 5% optimality gap in a given amount of time, extended by the number of instances solved to larger optimality gaps. Over the given relaxation, it is evident that column elimination with the different methods can work appropriately and be competitive with column generation.

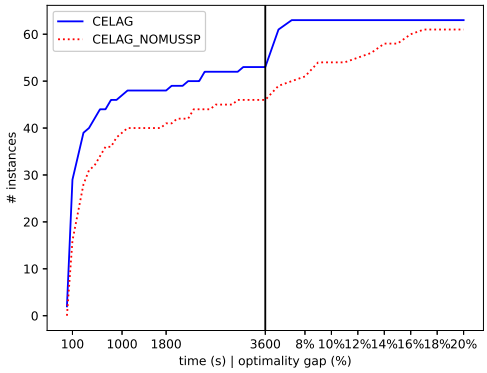
Evaluating the Impact of Cuts. We compare solving column elimination using CPLEX with and without cuts (CELP_CUT, CELP) and using the Lagrangian method with and without cuts (CELAG_CUT, CELAG). Figure 3.4.b is a performance plot for solving the instances up to 5% as in the last experiment. Figure 3.4.b shows how cuts greatly improve column elimination when using CPLEX as the LP solver, and benefit when solving the Lagrangian reformulation but not as much.

We then compare the performance of CELP_CUT with one class of cuts removed at a time: without the rounded capacity cuts (CELP_NORCC), without the strengthened comb inequalities (CELP_NOCOMB), and without the clique inequalities (CELP_NOCLIQUE). Figure 3.5.a is a performance plot of the number of instances solved to within a 1% optimality gap. Rounded capacity cuts provide the most benefit, the overhead of strengthened comb inequalities sometimes outweigh their benefit but not entirely if we more closely examine the bounds achieved for each instance, and clique inequalities can provide some benefit later in the method when it is able to be distinguished from separations and other cuts.

Evaluating the Impact of Arc Fixing. We consider the impact of arc fixing by removing the feature from CELP_CUT to get CELP_CUT_NOFIX. Figure 3.5.b is a performance plot using 1% optimality gap. Arc fixing speeds up column elimination to find stronger bounds in less time.

Evaluating the Impact of muSSP. We evaluate the impact of using the muSSP algorithm to solve the subproblem in CELAG by removing it in CELAG_-NOMUSSP. The performance plot using 5% optimality gap is for 64 large X instances and shows that there is a significant speedup. We chose to use the X class here because the speedup is more pronounced on large instances.

Comparison to State-of-the-Art. Figure 3.6.b compares the state-of-the-art BCP method's



a. Evaluating muSSP for CELAG

Class	NP	Pecin Gap (%)	CE Gap (%)
A	22	0.36	0.66
B	20	0.14	0.61
E-M	12	0.33	2.60
F	3	0.00	16.41
P	24	0.42	0.85
X	100	0.44	2.13

b. Comparing root node bounds

Figure 3.6: (a) A comparison of column elimination using the Lagrangian reformulation with and without the muSSP algorithm. (b) Comparing the root node lower bounds from Pecin et al. (Pecin et al. 2017b) (Pecin) and the lower bounds from column elimination (CE); both methods include cuts.

root node lower bounds (Pecin) with the best column elimination method settings that we chose through experimentation (CE). For each class of problems the table gives the number of problems in the class (NP) and the average optimality gap found at the root node over all instances. Pecin takes less than 3600 seconds to compute its bounds for all instances except the X class where it can take several hours. CE gaps are computed based on 3600 second runs for all classes except M, F, and X which are given 7200 seconds. The state-transition graph did not compile for 12 X instances, so we leave these out of the analysis. One F instance with large capacity resulted in a large diagram and 27% gap that can be reduced with more runtime. The better of the CELAG and CELP results is used; most small instances use CELP while large instances like almost all of the X class use CELAG. We also remove two E class instances with unconventional demand formatting.

3.9 Conclusion

We introduced a column elimination procedure for the capacitated vehicle routing problem (CVRP). Our methods works with a relaxed set of routes that are compactly represented in a state-space relaxation, and from which infeasible routes are iterative removed. We showed how we can use existing route relaxations for the CVRP, such as the q-route and ng-route relaxation, to compile good initial state-space relaxations. When the dynamic program contains exactly all of the feasible routes, we showed that a solution to the CVRP can be found by solving an arc flow formulation over the state-transition graph of the dynamic program. When a state-space relaxation is used, this model yields a dual bound. To strengthen the linear programming relaxation of our model we added valid inequalities; in particular, we showed how a class of clique cuts can be derived from the structure of the state-transition graph. To solve the model more efficiently, we considered solving a Lagrangian dual formulation for which we implemented a specialized successive shortest paths algorithm. In our experimental results, we demonstrated that column elimination is a viable alternative to column

generation for the CVRP, although the best known dual bounds from the literature, obtained by column generation with cutting planes, are generally stronger.

Chapter 4

Column Elimination for Arc Flow Formulations

This chapter presents column elimination as a general framework for solving an arc flow formulation. It is based on the work by Karahalios and van Hoeve (2024). This work introduces the notion of a relaxed dynamic program which is used to develop a general conflict refinement algorithm. Additionally, this work proposes a method for embedding column elimination in a branch-and-bound framework. Finally, an experimental evaluation shows that column elimination can be competitive with or outperform state-of-the-art methods on various problem domains. Specifically, we find that column elimination 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 six open instances of the pickup-and-delivery problem with time windows.

4.1 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* 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, we propose to represent a *relaxation* of the columns. While this may seem counterintuitive, 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 (2020a, 2022) as an alternative to branch-and-price to solve the graph coloring problem; Karahalios and van Hoeve (2022) present 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 (2023b) apply 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 integer programming problems, incorporating and formalizing the existing approaches. The formalization includes the introduction of *relaxed dynamic programs* and a novel conflict refinement algorithm based on this definition. We show how to embed column elimination in a branch-and-bound framework. 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 various problem domains, as it 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 six 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 4.2. We then present the general discrete optimization problem setting to which we apply column elimination in Section 4.3. In Section 4.4, we describe the underlying model of column elimination, combining dynamic programming and integer programming. Section 4.5 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 4.6. Section 4.7 presents the three combinatorial problems that we use as a computational case study. The experimental results are presented in Section 4.8. We provide a summary and conclusion in Section 4.9.

4.2 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 (Desrosiers et al. 1984, 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), and vehicle routing problems (Fukasawa et al. 2006,

Baldacci et al. 2011b, Pecin et al. 2017b, 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 4.8 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 4.4.

Another related approach is that of *arc flow formulations* for integer programming (de Lima et al. 2022). Arc flow formulations have been used successfully to model problems over directed networks with solutions that are either a single path (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. 2016b, Ciré and van Hoeve 2013, Bergman and Ciré 2018); we refer to Castro et al. (2022) and van Hoeve (2024) for recent surveys. In the previous works on column elimination, arc flow formulations were described using decision diagrams. This work instead uses state-transition graphs of dynamic programs to describe its networks, which are closely related to weighted decision diagrams (Hooker 2013) but offer a more generic modeling environment.

Column elimination works by solving iteratively strengthened discrete relaxations. Similar methods have been proposed for arc flow formulations, including iterative aggregation and disaggregation (Clautiaux et al. 2017), dynamic discretization discovery (Boland et al. 2017), and in the context of column generation, decremental state-space relaxation or state-space augmentation (Righini and Salani 2008, Boland et al. 2006). Column elimination differs from these methods by the flexibility in its initial relaxations, its refine algorithm and the use of a Lagrangian method to simultaneously refine and solve the arc flow formulation.

4.3 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. 2016b). 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:

$$(\mathcal{P}) \quad \min_{X \subseteq \mathcal{S}} \left\{ \sum_{x \in X} f(x) : C(X) = 1 \right\} \tag{4.1}$$

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. 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. For example, consider the CVRP from Chapter 3.

Example 4.3.1. 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 $x \in X$.

Also, 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. $U = \{\min_{i \in [n]} \ell_i, \min_{i \in [n]} \ell_i + 1, \ldots, \max_{i \in [n]} u_i\}$. 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 and 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.

4.4 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.

4.4.1 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. The simplest but largest model would have one arc for each sequence with its relevant costs, so choosing an appropriate state-space is what makes the dynamic program useful.

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).

4.4.2 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.

$$F: \quad \min \sum_{a \in \mathcal{A}} c(a) y_a \tag{4.2}$$

$$\text{s.t.} \sum_{a \in \mathcal{A}} g_j(a) y_a \ \circ_j \ b_j \qquad \forall\ j \in \{1, \ldots, m\} \tag{4.3}$$

$$\sum_{a \in \delta^+(s)} y_a - \sum_{a \in \delta^-(s)} y_a = 0 \qquad \forall\ s \in \mathcal{N} \backslash \{r, t\} \tag{4.4}$$

$$y_a \in \mathbb{Z}_+ \qquad \forall\ a \in \mathcal{A} \tag{4.5}$$

The objective (4.2) is to minimize the sum of the costs of the flows on arcs used in the solution. Constraints (4.3) are linear constraints using the additional cost functions from G with the comparators and right-hand side values from Γ. Constraints (4.4) 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 (4.5) are nonnegativity and integrality constraints. We prove the correctness of the model in Theorem 4.

Theorem 4. The optimal solution value of F is equal to the optimal solution value of $\mathcal{P}$.

Proof. 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 (4.4) and integrality constraints (4.5), 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), $\sum_{x \in X} f(x) = \sum_{a \in A'} c(a)$. □

4.4.3 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. 1981b). 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. 2016b). 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. We will present a generalized notion of a dynamic program relaxation, 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 (4.2)-(4.5) based on P'. Theorem 5 shows that F' is a relaxation of F.

Theorem 5. F' is a relaxation of F.

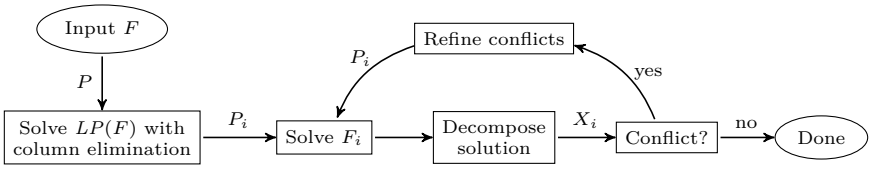


Figure 4.1: Pure column elimination for solving F.

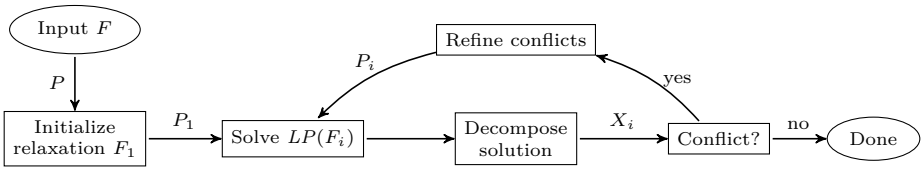


Figure 4.2: Column elimination for solving $LP(F)$.

Proof. 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. $\sum_{x \in X} f(x) \geq \sum_{x \in X} f'(x)$. □

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 CVRP model in Chapter 3.

4.5 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.

4.5.1 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 4.2. 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 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 4.6.

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 6.

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: $\quad\quad s_{curr} = s_{new}$

Theorem 6. 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 $x \notin \mathcal{S}_{i+1}$.

Proof. 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. □

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 7. Column elimination solves $LP(F)$ in a finite number of steps.

Proof. 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 6, 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 $LP(F)$. □

4.5.2 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.

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 4.1.

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 C. We show that column elimination solves F in Theorem 8.

Theorem 8. Column elimination solves F in a finite number of steps.

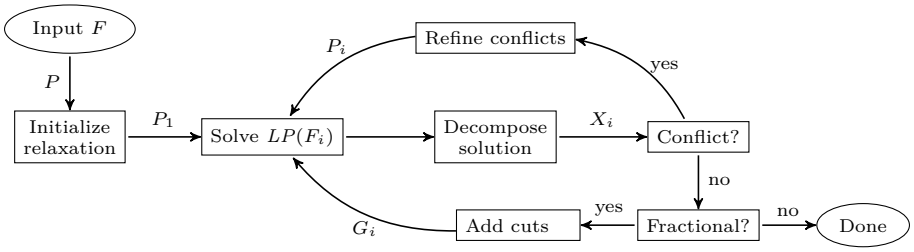


Figure 4.3: Cut-and-refine for solving F.

Proof. Proof Column elimination solves $LP(F)$ in a finite number of steps by Theorem 7. Then, by the same logic as the proof of Theorem 7, 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. □

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 4.3. When an optimal solution to $LP(F_i)$ 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 was shown to improve the performance of column elimination on some instances of the CVRP in (Karahalios and van Hoeve 2023b), which used problem-specific cuts. We prove the correctness and finite termination of cut-and-refine in Corollary 2, with the assumption that cuts are added from a finite cutting plane procedure.

Corollary 2. Cut-and-refine solves F in a finite number of steps.

Proof. 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. □

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 J 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 3.

Corollary 3. Branch-and-refine solves F in a finite number of steps.

Proof. 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. □

A natural extension is to embed a cutting plane procedure from Section 4.5.2 into the branch-and-bound search to strengthen the linear programming relaxations. This would yield a branch-cut-and-refine algorithm.

4.6 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) and the application-specific approach by Karahalios and van Hoeve (2023b).

4.6.1 Lagrangian Model

Recall that the linear programming relaxation $LP(F)$ contains constraints (4.3). Without loss of generality, we assume in this section that these constraints are of the following standard form:

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

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

$$L(\lambda): \quad \min \sum_{a \in \mathcal{A}} c_a y_a + \sum_{j=1}^{m} \lambda_j (b_j - \sum_{a \in \mathcal{A}} g_j(a) y_a) \tag{4.6}$$

$$\text{s.t.} \sum_{a \in \delta^-(u)} y_a - \sum_{a \in \delta^+(u)} y_a = 0 \qquad \forall u \in \mathcal{N} \setminus \{r, t\} \tag{4.7}$$

$$y_a \geq 0, y_a \in \mathbb{Z} \qquad \forall a \in \mathcal{A} \tag{4.8}$$

The objective function (4.6) can be rewritten as

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

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

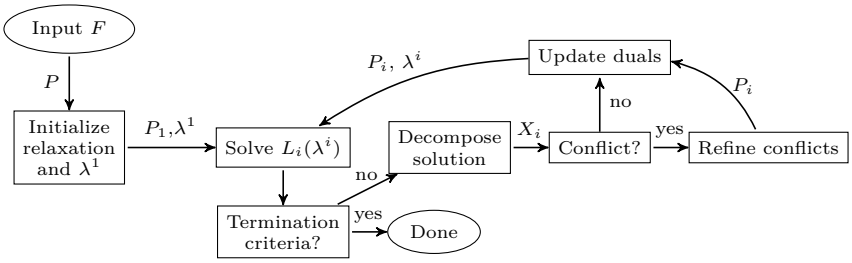


Figure 4.4: Column elimination for solving $LP(F)$ using subgradient descent.

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 E.

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 (2023b) for the CVRP, which also applies to the general case.

Proposition 3. For a fixed λ, $L(\lambda)$ can be solved in $O(K(|\mathcal{N}| \log(|\mathcal{N}|) + |\mathcal{A}|))$ time.

4.6.2 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 4.4. While the column elimination framework presented in Figure 4.2 solves $LP(F_i)$ at each iteration i, and then refines conflicts based on the optimal solution, the framework in Figure 4.4 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 D.

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. 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.

4.6.3 Cut-and-refine with Subgradient Descent

Modifying cut-and-refine to incorporate solving $LP(F)$ with subgradient descent is not straightforward. Indeed, in 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 (2023b) 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.

4.7 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 Chapter 3 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). 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.

4.8 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.

4.8.1 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. 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.

4.8.2 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 E-J. 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 E.)

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 F.)

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 G.)

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 I.)

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 J.)

4.8.3 Comparison with State-of-the-Art

We next compare column elimination to state-of-the-art algorithms for solving the VRPTW, the graph multicoloring problem, and the PDPTW. For each instance, we list the current best-known

upper bound (UB) that each algorithm takes as input. For each method, we report the lower bound (LB) and time taken (Time (s)). For branch-and-cut-and-price methods, we give the number of explored 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 cuts (Cuts), and the number of refinements (CR). For multicoloring instances, we also report the number of nodes n and edges m in the graph, an initial lower bound ω used for preprocessing, and the best upper bound found by each method.

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 called VRPSolver (Pessoa et al. 2020), which is based on branch-and-cut-and-price. We use VRPSolver with the default settings on the same server as the one we use for column elimination.

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 4.1 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 six instances. Table 4.2 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 show the full table of results for all instance classes in Appendix K.

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 full table of results are presented in Appendix L.

Table 4.1: 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	4085.95	29	150	992	0	3600
C2_4_10	3665.1	3647.88	1	3600	3397.41	1	175	2128	0	3600
C2_4_2	3914.1	3900.22	1	3600	3815.14	1	152	2153	0	3600
C2_4_3	3755.2	3723.96	1	3600	3348.42	1	79	1021	0	3600
C2_4_4	3523.7	3486.12	1	3600	2725.34	1	51	474	0	3600
C2_4_5	3923.2	3923.2	1	971	3831.85	1	376	5034	0	3600
C2_4_6	3860.1	3860.1	1	2466	3696.11	1	291	3892	0	3600
C2_4_7	3870.9	3870.9	1	1483	3692.25	1	253	3391	0	3600
C2_4_8	3773.7	3770.24	1	3600	3553.38	1	232	3090	0	3600
C2_4_9	3842.1	3806.45	1	3600	3568.74	1	210	2714	0	3600
C2_6_1	7752.2	7719.46	1	3600	7688.34	1	391	1671	0	3600
C2_6_10	7123.9	6340.81	1	3600	**6437.63**	1	94	1733	0	3600
C2_6_2	7471.5	7075.15	1	3600	**7177.06**	1	94	1546	0	3600
C2_6_3	7215	4670.06	1	3600	**5953.32**	1	41	593	0	3600
C2_6_5	7553.8	7540.44	1	3600	7241.6	1	231	4427	0	3600
C2_6_6	7449.8	7400.61	1	3600	6976.78	1	168	3227	0	3600
C2_6_7	7491.3	6294.69	1	3600	**6966.47**	1	151	2871	0	3600
C2_6_8	7303.7	7223.09	1	3600	6753.56	1	140	2559	0	3600
C2_6_9	7303.2	5754.15	1	3600	**6741.86**	1	104	1834	0	3600

Table 4.2: The performance of column elimination on three difficult VRPTW instances.

Instance		VRPSolver			Column Elimination				
Name	UB	LB	Nodes	Time (s)	LB	LPIt	LagIt	CR	Time (s)
C1_10_5	42434.8	42434.8	1	1227	42434.8	5	397	7468	6224
C1_8_5	25138.6	25138.6	1	737	25138.6	4	144	3198	1340
C2_10_1	16841.1	-	-	3600	**16841.1**	17	145	1661	10049

The results in Table 4.3 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).

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

Table 4.3: 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	LPIt	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**	141	581	139
COG-nsrand-ipx	13240	69510	30	-	-	3600	30	**30**	2	0	5
COG-opt1217	1536	6528	26	-	-	3600	26	**26**	2	0	13
COG-rout	560	2940	30	30	32	3600	30	**30**	2	0	0

are presented in Appendix M.

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 have reported results on many of the larger instances from Li and Lim (2001), we show in Table 4.4 six of these instances that column elimination closes.

Table 4.4: The performance of column elimination on six 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	15	217	50
LC1_4_5	7150.0	**7150.0**	9	32	609	477
LC1_4_6	7154.02	**7154.02**	19	73	1867	3295
LC1_6_5	14086.3	**14086.3**	8	91	2140	1620
LC2_2_1	1931.44	**1931.44**	37	54	494	300
LC2_4_1	4116.33	**4116.33**	12	113	1123	1555

4.9 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 six open instances of the pickup-and-delivery problem with time windows.

Chapter 5

Primal Heuristics for Arc Flow Formulations

This chapter proposes a novel primal heuristic for arc flow formulations and studies its performance in column elimination. Before this work, column elimination did not have a mechanism for finding feasible solutions at each iteration of its procedure. The proposed primal heuristic finds an initial solution by solving the arc flow formulation over a subgraph of the state-transition graph. Then, the heuristic tries to improve the solution with a large neighborhood search based on a destroy-and-repair approach. We embed the primal heuristic in column elimination and provide a computational evaluation on three variants of vehicle routing problems. The results show that the primal heuristic and large neighborhood search are effective for many instances.

5.1 Introduction

Primal heuristics are subroutines of exact methods that aim to find feasible solutions with good objective values. For integer programming, generic branch-and-bound solvers use several primal heuristics that are based on fractional solutions to linear programming relaxations. For large-scale integer programming, branch-and-price methods use restricted master heuristics and diving heuristics (Sadykov et al. 2019). So far, for arc flow formulations, exact methods are based on solving restricted arc flow formulations (Clautiaux et al. 2017) and greedy heuristics (van Hoeve 2022).

Large neighborhood search is a solution improving heuristic (Shaw 1998). Given an initial solution, the heuristic searches a neighborhood of related solutions. There are different ways to define a neighborhood that may depend on the modeling framework. Generic large neighborhood search methods exist for dynamic programming (Ergun and Orlin 2006), decision diagram solvers (Gillard and Schaus 2022) and constraint programming (Mouthuy et al. 2012). This work is an extension of large neighborhood search for dynamic programming to arc flow formulations over dynamic programs, which is more complicated due to solutions being collections of paths through the state-transition graph that must collectively satisfy side-constraints.

Contributions. The first contribution is a generic primal heuristic for exact methods that solve arc flow formulations. This includes a novel large neighborhood search method. The second contribution is a computational evaluation of the primal heuristic in column elimination on three variants of

vehicle routing problems.

The remainder of the chapter is structured as follows. Section 5.2 describes the primal heuristic. Section 5.3 embeds the primal heuristic in column elimination. Section 5.4 gives experimental results and Section 5.5 concludes the chapter.

5.2 Primal Heuristic

This section introduces the primal heuristic for arc flow formulations that is based on adaptive large neighborhood search. The input to the primal heuristic is a subset of feasible sequences in $\mathcal{S}$, which is equivalent to a set of paths through the state-transition graph and induces a subgraph of the state-transition graph. The primal heuristic finds an initial solution by solving the arc flow formulation restricted to this subgraph. Then, the primal heuristic tries to improve this solution with a large neighborhood search, where a neighborhood is defined as a subgraph of paths similar to those in the current solution.

The primal heuristic is given as pseudocode in Algorithm 2. ILine 1 finds an initial solution by solving the arc flow formulation over the restricted set of sequences. This can be done with an off-the-shelf integer linear programming solver. Line 2 sets the initial neighborhood size, denoted by parameter k. Line 3 checks if the heuristic should continue, which can be based on total time, time without an improvement, or if the initial 'solveSubMIP' cannot find an initial feasible solution. Line 4 runs the large neighborhood search given the current solution. Line 5 checks if an improving solution has been found. If so, this becomes the current solution and the neighborhood size is set back to the original value in Line 6. Otherwise, the size of the neighborhood is increased in Line 8.

Algorithm 2 Primal Heuristic

Input: An arc flow formulation F for a dynamic program P, a subset of sequences R
Output: A feasible solution X

```
1: X = solveSubMIP(F, R)
2: k = 1
3: while shouldContinue() do
4:    X' = largeNeighborhoodSearch(F,X,k)
5:    if f(X') < f(X) then
6:       X = X', k = 1
7:    else
8:       k = k + 1
9: return X
```

We propose a large neighborhood search method that uses a destroy-and-repair approach. The algorithm is given as pseudocode in Algorithm 3. Line 1 removes some of the elements from sequences in the solution. Line 2 constructs a neighborhood that considers reinserting the removed elements into the sequences or using sequences of only removed elements. Line 3 solves the arc flow formulation restricted to the neighborhood. There are common removal strategies for large neighborhood search methods that solve problems with solutions that are multiple sequences. The common strategies are to remove elements randomly, based on the worst contribution to the objective function, or based on similarities of the dynamic programming states at the heads of the transitions. Next, we describe the generic yet specific way of constructing a neighborhood based on the removed elements.

Algorithm 3 Large Neighborhood Search

Input: An arc flow formulation F for a dynamic program P, a solution X, a parameter k
Output: A feasible solution X

1: E = removeElementsFromSequences(X, k)
2: D_X = constructSolutionNeighborhood(P, X, E)
3: X = solveSubMIP(F, D_X)
4: **return** X

The procedure for constructing a neighborhood is given as Algorithm 4. Line 1 begins a loop that constructs a neighborhood for each sequence individually, which are then combined to create the output neighborhood. Line 2 creates the destroyed sequence and Line 3 constructs all sequences that involve inserting one or more elements in E into this destroyed sequence. Line 4 aggregates the neighborhoods, denoted as a union of subgraphs. The implementation of neighborhood constructions uses the dynamic program to efficiently build the subgraphs; this includes feasibility checks and the use of dominance rules. Proposition 4 gives an upper bound on the size of the neighborhood, although in practice many of these sequences are infeasible which makes the construction more efficient.

Algorithm 4 constructSolutionNeighborhood

Input: A dynamic program P, a set of sequences X, a subset of elements $E \subseteq U$
Output: A set of sequences N_X

1: **for** $x \in X$ **do**
2: $\quad x'$ = destroyedSubsequence(x, E)
3: $\quad D_{x'}$ = constructInsertionNeighborhood(P, x', E)
4: $\quad D_X = D_X \cup D_{x'}$
5: **return** D_X

Proposition 4. For each sequence $x \in X$, the number of sequences in the neighborhood is upper bounded by $\binom{|E|+1}{|x|} 2^{|E|} |E|!$.

Proof. Any subset of the items in E can be inserted into the sequence, with any permutation. An upper bound on this value is $2^{|E|}|E|!$. The sequence can be inserted into each permutation, maintaining its fixed order. So, an upper bound on the total ways to do this is $\binom{|E|+1}{|x|}$. □

5.3 Primal Heuristic in Column Elimination

This section describes how to embed the primal heuristic in column elimination. This requires column elimination to maintain a subset of feasible sequences as input to the primal heuristic. It also requires deciding when the primal heuristic should be run and for what duration. We give an overview of column elimination and consider these implementation details.

Figure 5.1 shows the steps of column elimination for solving the linear program relaxation of the arc flow formulation, $LP(F)$, including the primal heuristic. The primal heuristic can be embedded similarly in column elimination with subgradient descent. The algorithm initializes a restricted set of sequences R_1, and then adds the sequences found in the decomposition X_i at each iteration i. These sequences may be infeasible because they come from a solution to a relaxation of the arc flow

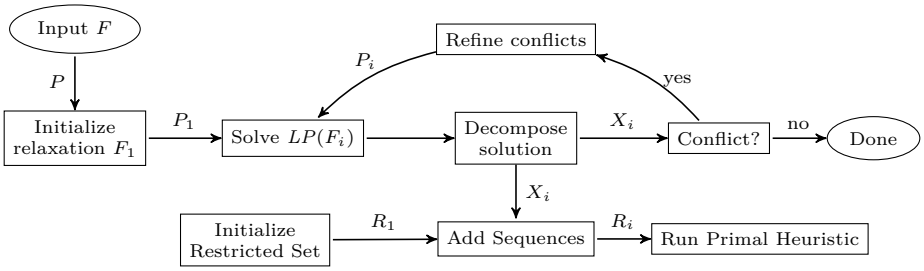


Figure 5.1: Column elimination for solving $LP(F)$, including the primal heuristic.

formulation F_i defined over a relaxed dynamic program P_i. In the experimental results, we consider adding both feasible sequences and truncated versions of infeasible sequences. We also consider rearranging elements within a sequence before adding the sequence to the restricted set, and refer to this as an *intrasequence swap*.

There is a tradeoff to using a primal heuristic in column elimination. While it may improve the current feasible solution, it uses time that could be allotted to the other components of the column elimination algorithm. We decide to run the primal heuristic whenever a new sequence is added to the restricted set, i.e. when $R_i \supset R_{i-1}$. The primal heuristic is terminated when an improvement has not been made for t seconds, and the large neighborhood search is run only when the initial restricted arc flow formulation finds an improving solution.

5.4 Experimental Evaluation

This section evaluates the performance of the primal heuristic on three types of problems: the capacitated vehicle routing problem, the vehicle routing problem with time windows, and the pickup and delivery problem with time windows. We use column elimination as our solution framework with the algorithmic settings as in (Karahalios and van Hoeve 2023a). In particular, we use column elimination with subgradient descent and change to solving the arc flow formulations with CPLEX when variable fixing sufficiently reduces the size of the state-transition graph. We study key implementation decisions and compare with the state-of-the-art.

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. For each experiment, we give each algorithm a timeout of 3,600 seconds. We consider two metrics for the performance of the primal heuristic. The primal gap compares a solution X to the best known solution X^* for an instance; it is computed as $\frac{f(X)-f(X^*)}{max(f(X^*),f(X))}$ Berthold (2013). The relative gap measures how much the best solution found can possibly be improved. It is based on the dual bound LB and the best upper bound UB found by an exact method and is computed as $\frac{UB-LB}{UB}$.

5.4.1 Impact of Implementation Decisions

We study the impact of several implementation decisions on the performance of the primal heuristic in column elimination. The implementation decisions are the removal strategy, the time limit, which sequences to add to the restricted set, and the large neighborhood search. The experiments in this subsection are based on instances of the vehicle routing problem with time windows from a set of

benchmark instances (Homberger and Gehring 1999), restricted to instances of types $C1$ and $C2$ that have 200 and 400 locations.

The removal strategy in the large neighborhood search directly affects the resulting neighborhoods that are searched. We consider three commonly used methods for choosing which elements to remove. First, a random choice. Second, the elements that contribute the most to the objective function. Third, the elements that are similar to each other as implemented in (Ropke and Pisinger 2006). We ran experiments with each removal strategy and a five second time limit. On average, the primal gap and relative gap for the three methods were as follows: for random, 0.169 and 0.201; for the second strategy, 0.281 and 0.305; for the third strategy, 0.209 and 0.246. So, the random strategy performed the best.

The time limit for the primal heuristic has a tradeoff between taking time from the exact method to explore more and larger neighborhoods. We consider three different values for the time limit for which the primal heuristic stops after not finding an improving solution for this much time. The values are 5 seconds, 30 seconds, and 60 seconds. We ran experiments with the random removal strategy and each time limit. The performance after 300 seconds of runtime is different than for the full 3600 seconds. Namely, after 300 seconds, the average primal gap and relative gap for the three time limits were as follows: for 5 seconds, 0.27 and 0.348; for 30 seconds, 0.306 and 0.411; for 60 seconds, 0.353 and 0.466. So, the 5 second timeout performed best. However, after 3600 seconds, the average primal gap and relative gap for the three time limits were as follows: for 5 seconds, 0.189 and 0.216; for 30 seconds, 0.181 and 0.23; for 60 seconds, 0.186 and 0.216. So, the longer time limits had some minor benefits later in the runtime. This suggests that exploring larger neighborhoods takes time, but can find better solutions than only exploring smaller neighborhoods.

The method for adding sequences to the restricted set is interesting because some of the sequences found in the decompositions during column elimination may be infeasible. We consider the impact of adding a feasible sequence created by truncating each infeasible sequence. We ran experiments with a random removal strategy and 5 second time limit, with ablations of the truncated sequences and intrasequence swaps. The results indicate that both ablations did not significantly affect the performance, suggesting that they are not major drivers of performance.

The large neighborhood search is intended to improve the solutions found by the restricted arc flow formulation. We ran experiments with column elimination without the primal heuristic, with the primal heuristic but without the large neighborhood search, and with the primal heuristic and the large neighborhood search. The experiments use the random removal strategy and a 5 second time limit. The results in Table 5.1 show that the primal heuristic, and particularly the large neighborhood search, are impactful.

Table 5.1: The average primal gap and average relative gap achieved by column elimination with and without the primal heuristic and with and without the large neighborhood search.

	CE		CE+MIP		CE+MIP+LNS	
	primal gap	rel gap	primal gap	rel gap	primal gap	rel gap
C1	0.824	0.826	0.031	0.04	0.014	0.024
C2	0.878	0.888	0.681	0.698	0.325	0.377

5.4.2 Comparison with State-of-the-Art

We analyze the performance of the primal heuristic in column elimination against state-of-the-art heuristics and exact methods. The solver PyVRP(Wouda et al. 2024), which is based on state-of-the-art heuristics, is used as a comparison for the capacitated vehicle routing problem and the vehicle routing problem with time windows. The solver VRPSolver (Pessoa et al. 2020), a state-of-the-art exact method for vehicle routing problems, is also used as a comparison (Pessoa et al. 2020) for these problems. We use the parameter settings 'CallFrequencyOfRestrictedMasterIpHeur = 1', 'MIPemphasisInRestrictedMasterIpHeur = 1', 'DivingHeurUseDepthLimit = 1', and 'CallFrequencyOfDivingHeur = 1' for VRPSolver. For the pickup-and-delivery problem with time windows, we use a fixed cost of $10,000$ for each vehicle which effectively changes the objective to use the least number of vehicles, which makes this problem more like a bin packing problem. For this problem, a comparison is drawn with a state-of-the-art heuristic from (Ropke and Pisinger 2006). We forego a comparison with an exact method, as we use large-scale instances that most solvers have not provided results for and for which VRPSolver does not generate bounds after the time limit.

The experiments for the capacitated vehicle routing problem use the set of benchmark 'X' instances from (Uchoa et al. 2017). The results in Table 5.2 show that the exact methods are far behind PyVRP. The primal heuristic in column elimination with VRPSolver both give decent solutions for instances with 100 locations. The primal heuristic in column elimination performs ok on instances with 200 locations, better than VRPSolver likely because it only runs a primal heuristic after solving the root node of column generation, which it often fails to do within the time limit.

Table 5.2: The average primal gap and average relative gap achieved for solving capacitated vehicle routing problems with column elimination, PyVRP, and VRPSolver. The results are aggregated by instances with different amounts of locations: $100-200$, $200-300$, $300-500$, and $600-1000$.

	CE		PyVRP		VRPSolver	
	primal gap	rel gap	primal gap	rel gap	primal gap	rel gap
X100	0.104	0.13	0.0	1.0	0.063	0.063
X200	0.147	0.184	0.002	1.0	0.599	0.598
X300-500	0.205	0.26	0.003	1.0	0.799	0.795
X600-1000	0.492	0.586	0.008	1.0	0.955	0.949

The experiments for the vehicle routing problem with time windows use the set of benchmark instances from Homberger and Gehring (1999). The results in Table 5.3 show that the primal heuristic can give quite good solutions for the $C1$ instances, which include instances with between 100 and 1000 locations. The $C1$ instances have tighter time windows and are usually easier to solve than the $C2$ instances. It does not perform nearly as well as PyVRP (which has some negative values because of rounding distances to integer values), but the solutions are still strong for this type. The primal heuristic does not perform as well for the other instance classes. VRPSolver encounters the same issue as before, where it does not run a primal heuristic before finishing column generation.

The experiments for the pickup-and-delivery problem with time windows use the set of benchmark instances from (Li and Lim 2001). This problem has two objectives. The first is to minimize the number of vehicles used. The second is to minimize the total travel time of the vehicles used. As many heuristics for this multi-objective problem use an approach that removes an entire sequence and

Table 5.3: The average primal gap and average relative gap achieved for solving vehicle routing problems with time windows with column elimination, PyVRP, and VRPSolver. The results are aggregated by instances of types $C1$, $C2$ which are clustered instances and then separately the R and RC instances which involve at least some randomness.

	CE		PyVRP		VRPSolver	
	primal gap	rel gap	primal gap	rel gap	primal gap	rel gap
C1	0.057	0.093	-0.002	1.0	0.421	0.421
C2	0.414	0.48	-0.002	1.0	0.709	0.711
R	0.723	0.793	0.003	1.0	0.764	0.765
RC	0.69	0.782	0.001	1.0	0.795	0.797

tries to redistribute its locations, we implement an additional destroy strategy that tries removing each sequence in addition to k random elements. The results in Table 5.4 show again that the heuristic method clearly outperforms the primal heuristic in column elimination. However, again, the primal heuristic finds good solutions for certain instances such as the $C1$ instances with 200 locations. For the other three instances types, the removal strategy of destroying an entire sequence is beneficial.

Table 5.4: The average primal gap and average relative gap achieved for solving pickup and delivery problems with time windows with column elimination and (Ropke and Pisinger 2006). The results are aggregated by instance type and number of locations.

	CE Random		CE Random + Sequence		Ropke & Pisinger	
	primal gap	rel gap	primal gap	rel gap	primal gap	rel gap
C1 200	0.14	0.256	0.164	0.281	0.005	1.0
R1 200	0.34	0.559	0.28	0.537	0.019	1.0
C1 400	0.22	0.438	0.208	0.426	0.008	1.0
R1 400	0.493	0.73	0.413	0.703	0.036	1.0

5.5 Conclusion

This chapter introduced a primal heuristic for arc flow formulations and embedded it into column elimination. A positive aspect of the method is that it is generic. The primal heuristic showed good performance on instances of vehicle routing problems that had 100 or 200 clustered locations. The results revealed that the primal heuristic can achieve decent solutions before state-of-the-art column generation methods terminate, and suggest that it may be worth further investigating primal heuristics for arc flow formulations.

Chapter 6

Cutting Planes for Arc Flow Formulations from Path Flow Formulations

This chapter proposes a novel method for translating cutting planes of path flow formulations to arc flow formulations. There exist cutting planes for path flow formulations that cannot be represented in the equivalent arc flow formulations. This work shows how an arc flow formulation can be altered to permit such cuts. We give an example of the method for adding subset-row cuts to arc flow formulations, which are classified as non-robust in the column generation literature. Implementing these cuts greatly improved branch-and-price solvers for vehicle routing problems. We implement the method in column elimination and give experimental results for adding subset-row cuts to the arc flow formulation that demonstrate its performance benefits.

6.1 Introduction

An arc flow formulation has an equivalent path flow formulation, which has a variable for each path through the state-transition graph instead of for each arc. The path flow formulation is often solved with branch-and-price using column generation because of the large number of variables. So, branch-and-cut-and-price uses cutting planes that are represented in terms of these path-based variables. These cutting planes have improved the performance of branch-and-price methods. Robust cutting planes improve the performance without requiring changes to the pricing problem, while non-robust cutting planes can greatly improve the performance of branch-and-cut-and-price, but require being implemented in a weakened form (Pecin et al. 2017b). Recently, a third type of cutting plane was introduced, called a resource-robust cut, which use states in the dynamic program of the pricing problem to deduce coefficients that create weakened non-robust cuts (Hoogendoorn and Dalmeijer 2023). The goal of this work is to translate cutting planes that use path-based varaiables into cutting planes for arc flow formulations.

Cutting planes have been successfully added to column elimination in a previous work (Karahalios and van Hoeve 2024). However, the work only effectively added cutting planes that are equivalent to robust cutting planes in column generation. Some non-robust cuts were formulated, but the

implementation did not show performance improvements. Further, the robust cutting planes were difficult to implement in column elimination with subgradient descent, which is a common issue in relax-and-cut methods (Lucena 2005). This work gives a method for effectively adding non-robust cuts to column elimination, but does not yet column elimination with subgradient descent.

Contributions. The first contribution is a novel method to translate a cutting plane from a path flow formulation into an arc flow formulation. This includes non-robust cuts. The second contribution is an implementation of subset-row cuts in column elimination. The third contribution is a computational evaluation on instances of the capacitated vehicle routing problem.

The remainder of the paper is structured as follows. Section 6.2 gives preliminary definitions and notation for arc flow formulations and path-based formulations, Section 6.3 discusses translating cutting planes from the path flow formulation into the arc flow formulation, Section 6.4 shows experimental results, and Section 6.5 concludes the paper.

6.2 Arc Flow Formulations and Path Flow Formulations

In this section, we recall the general arc flow formulation from Karahalios and van Hoeve (2024) and present an equivalent path flow formulation.

Arc Flow Formulation. The general form of the arc flow formulation is a constrained network flow problem over a state-transition graph D. For each arc $a \in \mathcal{A}$, there is a decision variable y_a representing the 'flow' along that arc. The model is:

$$
\begin{aligned}
F: \quad \min \ & \sum_{a \in \mathcal{A}} c(a) y_a && && (6.1)\\
\text{s.t.} \ & \sum_{a \in \mathcal{A}} g_j(a) y_a \ \geq \ b_j && \forall\ j \in \{1, \dots, m\} && (6.2)\\
& \sum_{a \in \delta^+(s)} y_a - \sum_{a \in \delta^-(s)} y_a = 0 && \forall\ s \in S \backslash \{r, t\} && (6.3)\\
& y_a \in \mathbb{Z}_+ && \forall\ a \in A. && (6.4)
\end{aligned}
$$

Constraints (6.3) represent flow conservation at each node. The constraint set (6.2) are used to represent side constraints, which use cost functions from the set $G = \{g_j\}_{j=1}^{|J|}$ from the underlying dynamic program.

Path Flow Formulation. The path flow formulation has an integer variable for each $r - t$ path in D. Let Θ be the set of paths through D and let each path $\theta \in \Theta$ be an ordered set of arcs along the path. The model is:

$$
\begin{aligned}
F': \quad \min \ & \sum_{\theta \in \Theta} (\sum_{a \in \theta} c(a)) z_\theta && && \\
\text{s.t.} \ & \sum_{\theta \in \Theta} (\sum_{a \in \theta} g_j(a)) z_\theta \geq b_j && \forall j \in \{1, 2, \dots, m\} && (6.5)\\
& z_\theta \in \mathbb{Z}_+ && \forall \theta \in \Theta. &&
\end{aligned}
$$

Define $\phi : \mathbb{R}^{|\Theta|} \to \mathbb{R}^{|\mathcal{A}|}$ as a projection of feasible solutions to the path flow formulation into feasible solutions to the arc flow formulation. Each path is a unique sequence of elements, so this projection is uniquely defined. Let $Sol(F)$ be the set of feasible solutions to a formulation F and

$Opt(F)$ the optimal solution value. The proof follows from a flow decomposition and the additive structure of the coefficients.

Proposition 5. $\{\phi(z) : z \in Sol(F')\} = Sol(F)$ and $Opt(F') = Opt(F)$

6.3 Translating a Cutting Plane from a Path Flow Formulations to an Arc Flow Formulation

Let $\alpha \in \mathbb{R}^{|\Theta|}$, $\beta \in \mathbb{R}$. A cutting plane for the path flow formulation has the form:

$$\sum_{\theta \in \Theta} \alpha_\theta z_\theta \geq \beta \tag{6.6}$$

An equivalent cutting plane in the arc flow formulation has coefficients $\gamma \in \mathbb{R}^{|\mathcal{A}|}$ such that $\sum_{a \in \theta} \gamma_a = \alpha_\theta$ for all $\theta \in \Theta$. It is not always straightforward to find such coefficients. In fact, because each arc can be part of many paths, such coefficients may not exist. However, we show that it is possible to modify the structure of the state-transition graph (via updating its underlying dynamic program) so that appropriate coefficients exist for the resulting arc flow formulation. To do this, we assume α is non-negative.

The modification method relies on the following observation. The cutting plane can be considered as an additional cost function to be appended to G. So, it is still possible to write a dynamic program P' that encodes the same sequences and costs as P, but also has these additional costs. Then, P can be seen as having a cost of zero for all transitions for this additional cost function. Because α is non-negative, P has the same set of sequences as P', but with the additional cost of each sequence in P being bounded by the cost in P'. This means that P is a dynamic program relaxation of P', which means that the conflict refinement algorithm from that work can be used to update the additional costs for paths with non-zero coefficients (Karahalios and van Hoeve 2024).

Algorithm 5 is pseudocode that summarizes the modification method. Let $\Theta_{>0}$ be the set of paths with non-zero coefficients in the path-based cutting plane. Line 1 copies the set of additional cost functions. Line 2 updates the dynamic program to have an additional cost with all zero values. Line 3 defines the dynamic program that includes the correct additional costs for the cutting plane. Line 5 loops over each path $\theta \in \Theta_{>0}$. Line 6 updates the dynamic program relaxation so that the arcs along the path have correct additional costs Karahalios and van Hoeve (2024). Proposition 6 proves the correctness of the algorithm and Proposition 7 shows how the modification algorithm affects the size of the formulation.

Let $F_{\alpha,\beta}$ be the arc flow formulation over P_α created by running Algorithm 5 and $F'_{\alpha,\beta}$ the path flow formulation with the cutting plane added.

Proposition 6. $\{\phi(z) : z \in Sol(F'_{\alpha,\beta})\} = Sol(F_{\alpha,\beta})$ and $Opt(F'_{\alpha,\beta}) = Opt(F_{\alpha,\beta})$

Proof. For each path $\theta \in \Theta_{>0}$, the conflict refinement algorithm returns a new dynamic program relaxation that contains the same set of paths as the original dynamic program, and all paths have the same costs except θ, which has the costs of P_α. So, for each $\theta \in \Theta$, $\sum_{a \in \theta} g_{j+1}(a) = \alpha_\theta$. Thus, the arc flow formulation could only remove solutions that were violated by the cutting plane and the solution values remain the same. □

Algorithm 5 Modification Algorithm

Input: Dynamic program P based on f,$\mathcal{S}$, and G; cutting plane coefficients α; set of paths with non-zero coefficients $\Theta_{>0}$

Output: Updated dynamic program P_α

1: $G' = G$
2: $g_{j+1} = 0$, $G = G \cup g_{j+1}$
3: $g'_{j+1} = \alpha$, $G' = G' \cup g'_{j+1}$
4: P_α = formulateDynamicProgram(f,$\mathcal{S}$,G')
5: **for** $\theta \in \Theta_{>0}$ **do**
6: conflictRefinementAlgorithm(P_α,P,θ)
7: **return** P_α

Proposition 7. The modification algorithm increases the size of the formulation by at most $|\Theta_{>0}||U||K|$ where K is the length of the longest sequence in $\Theta_{>0}$.

Proof. Consider the number of arcs that can be added when the conflict refinement algorithm is applied to a single path. A node can be added for each element in the path, and each node can have at most U transitions. □

Example 6.3.1. Consider the capacitated vehicle routing problem from Chapter 3. Recall the dynamic program from that chapter, which had a set of additional cost functions $G = \{g_j\}_{j=1}^{|J|}$ for each location. Let $\mu \in \mathbb{R}^m$ be a vector of non-negative multipliers. Assume $0 \leq \mu \leq 1$. For the path flow formulation F', a Chvátal-Gomory cutting plane (also referred to as a subset-row cut (Jepsen et al. 2008)) is defined by the following vector of coefficients α and right-hand side β.

$$\alpha_\theta = \lfloor \sum_{j=1}^{m} \mu_j \sum_{a \in \theta} g_j(a) \rfloor \quad \beta = \lfloor \mu^T \mathbf{1} \rfloor \tag{6.7}$$

The dynamic program to represent routes in the capacitated vehicle routing problem can be updated with an additional cost function g_{j+1} to allow for the translation of the cutting plane. Let N_θ be the set of elements visited in θ. The coefficients depend only on the set of visited elements, because a route visits a location at most once. So, we define a coefficient for all routes with the same set of locations N.

$$\alpha_N = \lfloor \sum_{j \in N} \mu_j \rfloor \tag{6.8}$$

This allows us to define the additional cost function for each arc. The cost will be equal to 1 when the arc adds a location to the visited set that increases the coefficient and 0 otherwise. More formally, the cost function is $g_{j+1}(((\mathrm{NG}, w, v), i)) = 1$ if $\alpha_{\mathrm{NG}} = \alpha_{\mathrm{NG} \cup \{i\}} + 1$ and 0 otherwise.

Proposition 8. For each $\theta \in \Theta$, $\sum_{a \in \theta} g_{j+1}(a) = \alpha_\theta$.

Proof. Consider a particular θ. Let NG_0,...,NG_k be the NG sets at the states visited along θ, in order from root to terminal. By definition of the coefficient, $\sum_{a \in \theta} g_{j+1}(a) = \sum_{j=1}^{k} \alpha_{NG_j} - \alpha_{NG_{j-1}}$. Because $0 \leq \mu_j \leq 1$, $\alpha_{NG_j} - \alpha_{NG_{j-1}} \leq 1$. So, with $\alpha_\varnothing = 0$ and $\alpha_{N_\theta} = \alpha_\theta$, because $\alpha_{NG_j} - \alpha_{NG_{j-1}} \leq 1$, there must be $\alpha_{N_\theta} = \alpha_\theta$ arcs that contribute a value of 1 along the path θ. □

6.4 Experimental Results

We evaluate the modification method by implementing subset-row cuts in column elimination to solve the capacitated vehicle routing problem. We use column elimination as our solution framework with the algorithmic settings as in Karahalios and van Hoeve (2023a). In particular, we use column elimination with subgradient descent and switch to solving the linear programming relaxation of the arc flow formulation with CPLEX once variable fixing can substantially shrink the formulation. In the analysis, we consider instances where column elimination switches to using CPLEX, because it more clearly shows the benefit of the cutting planes, while an implementation that uses these cuts during column elimination with subgradient descent is for future work.

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. For each experiment, we give each algorithm a timeout of 3,600 seconds. We give as input the best known bound plus one. The relative gap measures how much the best solution found can possibly be improved. It is based on the dual bound LB and the best upper bound UB found by an exact method and is computed as $\frac{UB-LB}{UB}$. The experiments are for instances of the capacitated vehicle routing problem from (Augerat et al. 1998)

The experiments consider adding rounded capacity cuts ('RCC') by using the software package CVRPSEP (Lysgaard 2003) and three varieties of subset-row cuts, identified as useful ones in Jepsen et al. (2008). First, 'SRC3' cuts which have $\mu_j = \frac{1}{2}$ for three locations and 0 for the remaining locations. Second, 'SRC4' cuts which have $\mu_j = \frac{2}{3}$ for four locations and 0 for the remaining locations. And third, 'SRC5' cuts which have $\mu_j = \frac{1}{3}$ for five locations and 0 for the remaining locations. All possible subsets of locations are checked in order to identify violated cuts at each iteration. A maximum of 100 cuts are added at each iteration.

Table 6.2 shows the relative optimality gap for instances where column elimination solved a linear programming relaxation of the arc flow formulation and could not find any more cuts or separations; the relative optimality gap is computed based on this value, before switching to solving the arc flow formulation as an integer program. Some instances where the performance does not change much are also omitted to highlight the instances where adding cutting planes improves the performance. It should also be noted that cutting planes can impact variable fixing, which can affect the lower bound and the overall performance. The results indicate that the rounded capacity cuts often have the greatest impact, followed by the SRC3 cuts, and then marginal benefits for SRC4 and SRC5 cuts. This might be because the separation method takes too long to check for violated SRC4 and SRC5 cuts.

Table 6.1 gives the average the number of cuts added for each cut type, and the average difference in the number of arcs at the end of the method. The number of cuts is not too high, except for the SRC5 cuts, which could hinder performance. The size of the formulation grows by about 10%, which means that many sequences are involved in these cuts. This helps to show the tradeoff between the lower bound improvement and the possible slowdown from adding the cuts.

6.5 Conclusion

This chapter discussed a method for translating cutting planes from path flow formulations to arc flow formulations. To demonstrate the method, non-robust subset-row cuts from the column generation literature were translated in this way. The method relies on column elimination and the definition of a relaxed dynamic program. Experimental results show that subset-row cuts can improve lower bounds in column elimination when using CPLEX to solve the linear programming relaxation of the arc

Table 6.1: The average difference in the number of arcs, and the average number of each type of cut added.

	RCC	RCC + SRC3	RCC + SRC34	RCC + SRC345
Num Arcs Difference	-0.04	0.11	0.11	0.09
Num RCC Cuts	90	90	88	90
Num SRC3 Cuts	0	248	190	158
Num SRC4 Cuts	0	0	141	74
Num SRC5 Cuts	0	0	0	410

flow formulation, but the cutting planes were not yet added to column elimination with subgradient descent. This future work could unlock column elimination to solve many more instances, but requires some additional ideas and engineering.

Table 6.2: The relative optimality gap for each instance. 'SRC34' means that both SRC3 and SRC4 cuts are added. 'SRC345' means that SRC3, SRC4, and SRC5 cuts are added. Values in italics indicate that the column elimination stopped before the timeout because no more separations were possible and no more cuts were identified for the current solution.

	No Cuts	RCC	RCC + SRC3	RCC + SRC34	RCC + SRC345
A-n33-k6.vrp	*0.013*	*0.002*	*0.0*	0.0	0.0
A-n34-k5.vrp	*0.039*	*0.005*	*0.0*	*0.0*	*0.0*
A-n37-k5.vrp	*0.017*	0.005	*0.001*	*0.001*	0.002
A-n37-k6.vrp	*0.023*	*0.009*	0.005	0.005	0.005
A-n38-k5.vrp	*0.044*	*0.01*	0.006	0.006	0.006
A-n39-k5.vrp	*0.026*	*0.005*	0.003	0.002	0.001
A-n39-k6.vrp	*0.027*	0.009	0.004	0.004	0.004
A-n45-k6.vrp	*0.014*	0.003	0.001	0.001	0.001
A-n45-k7.vrp	*0.019*	0.004	0.002	0.001	0.003
A-n48-k7.vrp	*0.019*	0.002	*0.001*	*0.001*	*0.001*
A-n53-k7.vrp	*0.015*	0.007	0.004	0.005	0.005
A-n55-k9.vrp	*0.014*	0.004	0.002	0.002	0.002
B-n31-k5.vrp	*0.068*	0.079	0.079	0.076	0.075
B-n34-k5.vrp	*0.041*	0.005	0.005	0.005	0.004
E-n51-k5.vrp	*0.009*	0.007	0.003	0.003	0.005
P-n16-k8.vrp	*0.016*	*0.006*	*0.002*	*0.002*	*0.002*
P-n20-k2.vrp	*0.023*	*0.007*	*0.0*	0.0	0.0
P-n22-k2.vrp	*0.007*	*0.002*	0.0	0.0	*0.0*
P-n40-k5.vrp	*0.014*	*0.005*	0.0	0.0	*0.0*
P-n45-k5.vrp	*0.015*	0.007	0.008	0.006	0.008
P-n50-k10.vrp	*0.014*	*0.011*	*0.002*	*0.002*	*0.001*
P-n50-k7.vrp	*0.016*	0.008	*0.0*	0.0	*0.0*
P-n50-k8.vrp	*0.026*	*0.023*	0.015	0.015	0.018
P-n51-k10.vrp	*0.012*	*0.008*	*0.0*	0.0	0.0
P-n55-k10.vrp	*0.021*	*0.019*	0.01	0.009	0.01
P-n55-k15.vrp	*0.021*	*0.017*	*0.007*	*0.006*	*0.006*
P-n55-k7.vrp	*0.022*	0.017	0.014	0.015	0.025
P-n55-k8.vrp	*0.023*	0.015	0.015	0.014	0.014
P-n60-k10.vrp	*0.01*	*0.007*	*0.002*	*0.002*	0.001
P-n60-k15.vrp	*0.009*	*0.006*	*0.001*	*0.002*	*0.001*
P-n65-k10.vrp	*0.01*	*0.006*	0.002	0.0	0.004

Chapter 7

Variable Orderings in Column Elimination: A Portfolio Approach

This chapter studies how to choose which arc flow formulation(s) to use when applying column elimination to solve the vertex coloring problem. The original work on column elimination used binary decision diagrams as the state-transition graph for its arc flow formulations (van Hoeve 2022). There are many choices of binary decision diagrams to use to create the arc flow formulation for the vertex coloring problem, each having a different variable ordering. The experimental results showed that the choice of variable ordering can greatly affect the performance of column elimination. We propose a portfolio approach to selecting the best ordering among a set of alternatives. We consider several different portfolio mechanisms: a static uniform time-sharing portfolio, an offline predictive model of the single best algorithm using classifiers, a low-knowledge algorithm selection, and a dynamic online time allocator. As a case study, we compare and contrast their performance on the graph coloring problem. We find that on this problem domain, the dynamic online time allocator provides the best overall performance.

7.1 Introduction

Relaxed decision diagrams have recently been successfully applied within a range of solution methodologies for discrete optimization, including constraint programming, integer linear programming, integer nonlinear programming, and combinatorial optimization. For exact decision diagrams (e.g., reduced ordered binary decision diagrams), it is well known that the variable ordering greatly influences the size of the diagram (Bryant 1986, 1992, Wegener 2000). Likewise, for relaxed decision diagrams the variable ordering is often of crucial importance for their effectiveness. For example, Bergman et al. (Bergman et al. 2012a, 2014a) demonstrate that a variable ordering that yields a small exact diagram typically also provides stronger dual bounds from the relaxed diagram.

In some cases, e.g., for sequential scheduling problems, the variable ordering is prescribed by the sequential nature of the application. In most cases, however, we must design and/or select a variable ordering that we expect to perform well. In the literature several variable ordering strategies, generic as well as problem-specific, have been proposed. When decision diagrams are built from a single top-to-bottom compilation, dynamic variable orderings can be very effective. For example, a recent work

by Cappart et al. (Cappart et al. 2019) deploys deep reinforcement learning to dynamically select the next variable during compilation. Dynamic variable orderings are less applicable, however, to compilation via iterative refinement, in which case the ordering must be specified in advance. Oftentimes there is no single variable ordering strategy that dominates all others, and the challenge in practice is to select a strategy that works well for a specific instance. This is a well-studied problem in artificial intelligence, in the context of *algorithm portfolios*.

There are several ways to construct an algorithm portfolio: using static or dynamic features, formulating predictive models at the algorithm or portfolio level, predicting one algorithm to run per instance or creating a schedule of algorithms to run, using a fixed portfolio or updating it online (Kotthoff 2016). In this work, as we consider variable ordering strategies for relaxed decision diagrams, our goal is to study which portfolio design leads to the best performance of the diagram.

As a case study, we consider the graph coloring problem, for which a decision diagram approach was recently introduced (van Hoeve 2020b, 2022). It uses an iterative refinement procedure much like Benders decomposition or lazy-clause generation, by repeatedly refining conflicts in the diagram until the solution is conflict free. Our experimental results show several insights, at least for this problem domain: First, even the simplest portfolio (the static uniform time allocation) can already outperform all individual orderings. Second, predictive methods using classification models or exploration phases can lead to more instances solved optimally. However, these methods may lead to delayed optimality results on problem instances that are easy to solve. Third, allocating time to more than one variable ordering can yield a solution with a unique best upper bound from one ordering and a unique best lower bound from a different ordering. This indicates that it may be advantageous to use one variable ordering to obtain a lower bound and another to obtain the upper bound.

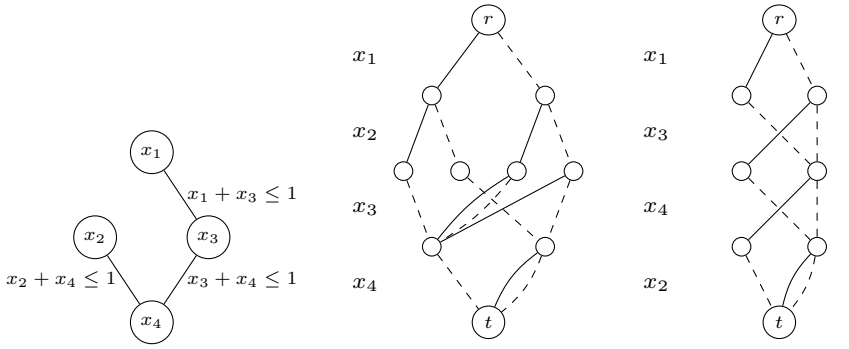
7.2 Decision Diagrams

We follow the framework of Bergman, Cire, van Hoeve, and Hooker (Bergman et al. 2016a) and introduce decision diagrams as a graphical representation of a set of solutions to a discrete optimization problem P defined on an ordered set of decision variables $X = \{x_1, x_2, \ldots, x_n\}$ and (optionally) an objective function $f(X)$ to be minimized or maximized.

7.2.1 Definitions

A *decision diagram* for P is a layered directed acyclic graph $D = (N, A)$ with node set N and arc set A. Diagram D has $n + 1$ layers of nodes, where a node in layer j represents a state associated with variable x_j. Layer 1 contains a single root node r, and layer $n + 1$ contains a single terminal node t. Arcs are directed from a node u in layer j to a node v in layer $j + 1$ and labeled with a decision value for variable x_j. The outgoing arcs for each node must have unique labels. Hence, an arc-specified r-t path $p = (a_1, a_2, \ldots, a_n)$ defines a complete variable assignment by setting x_j to be the label of a_j for $j = 1, \ldots, n$. We let $\text{Sol}(D)$ be the set of solutions represented by all r-t paths of D. We will slightly abuse notation and denote by $\text{Sol}(P)$ the set of feasible solutions to problem P. We say that D is an *exact* decision diagram for P if $\text{Sol}(D) = \text{Sol}(P)$. D is a *relaxed* decision diagram for P if $\text{Sol}(P) \subseteq \text{Sol}(D)$.

The objective function $f(X)$ can be represented in D by appropriately associating a 'weight' to each arc in the diagram. We define the weight of an r-t path as a function (e.g., the sum) of its arc weights, and require that the weight of the path is equal to the objective value of the solution



a. Constraint graph b. Lexicographic ordering b. Path ordering

Figure 7.1: Constraint graph (a) and exact decision diagrams using the lexicographic variable ordering (b) and the 'path' ordering (c), for the problem in Example 7.2.1. In the decision diagrams, dashed arcs represent arcs with label 0, while solid arcs represent arcs with label 1.

it encodes. The shortest (or longest) path in D can be computed in linear time since D is acyclic. Such path corresponds to an *optimal* solution if D is exact, and yields a *dual bound* if D is relaxed.

We can extend the application of decision diagrams to let *multiple* paths in D represent the solution to an optimization problem, as proposed in (van Hoeve 2020b, 2022). In that case, an optimal solution can be computed as a constrained network flow. We will use this application in our case study in Section 7.4.

7.2.2 Compilation Methods

We limit our discussion to the two most popular decision diagram compilation methods in the context of discrete optimization (Bergman et al. 2016a): top-down compilation and iterative refinement. Both methods rely on an underlying recursive formulation of the problem P, using states (associated with each node in N) and labeled transition functions (represented by the arcs in A).

Top-down compilation expands the diagram one layer at the time. It considers the nodes (states) in the previous layer, and creates all possible states according to the transition function. Equivalent states are merged. For relaxed decision diagrams, it is typical to impose a maximum size (or 'width') on the layers, in which case non-equivalent nodes may need to be merged. This compilation method can be applied recursively in a branch-and-bound like scheme to obtain an exact solution method.

Iterative refinement alternatively starts with an initial relaxed decision diagram in which each layer contains a single node, and all possible arcs between the nodes in subsequent layers are present. The diagram is then iteratively refined by splitting nodes and/or removing infeasible arcs. This is the method of choice for MDD-based constraint propagation, in which case refinement is again limited until a maximum width is reached. It can also be applied as a stand-alone exact solution method, by repeated computation of the optimal solution (which provides a dual bound) and refining any constraints that are violated along the optimal path(s).

7.2.3 Variable Ordering

As mentioned in Section 7.1, the variable ordering can have a crucial impact on the size of the decision diagram. This is illustrated in the following example:

Example 7.2.1. Consider the following constraint satisfaction problem:

$$x_1 + x_3 \leq 1, \quad x_2 + x_4 \leq 1, \quad x_3 + x_4 \leq 1,$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

The constraint graph for this problem is shown in Fig. 7.1(a). The associated exact decision diagram following the lexicographic variable ordering (x_1, x_2, x_3, x_4) is shown in Fig. 7.1(b). In Fig. 7.1(c) we show a smaller exact decision diagram using the path-ordering (x_1, x_3, x_4, x_2) that follows from the constraint graph.

Finding the variable ordering that yields the smallest exact decision diagram is an NP-hard problem (Wegener 2000). In practice, one therefore typically relies on heuristic variable ordering strategies. An example of a *problem-specific* variable ordering is the *maximal path decomposition* heuristic for compiling the independent sets of a graph (Bergman et al. 2012b, 2014a). It relies on an a priori computed path decomposition of the input graph, and selects the next variable according to this decomposition. An example of a *generic* variable ordering is the *k-look ahead* ordering (Bergman et al. 2012b, 2014a). It selects the variable that yields the smallest-width layer when $k = 1$, and evaluates a subset of k variables in general. We will present several more variable ordering heuristics for our case study in Section 7.4.

The maximal path decomposition heuristic is *static* as the ordering is determined once in advance. In contrast, the k-look ahead ordering is *dynamic* because the selection of the next variable is determined during the compilation and depends on the previous choices. Likewise, the reinforcement learning approach of Cappart et al. (Cappart et al. 2019) is a dynamic variable ordering heuristic by design. It uses an action-value function, based on neural fitted Q-learning, to determine the best variable to add to the ordering at each step. Due to its dynamic nature, it can however not be effectively applied when the decision diagram is compiled using iterative refinement (as in our case study). In our case study, we compose a portfolio of static orderings for settings where iterative refinement is used.

7.3 Algorithm Portfolio Design

Algorithm portfolios have been studied widely in artificial intelligence, and have been shown to be particularly effective for combinatorial optimization and Boolean satisfiability (Gomes and Selman 2001, Xu et al. 2008, Gagliolo and Schmidhuber 2011). While many variants exist, most approaches either select one algorithm among a set of alternatives to solve a given problem, or run multiple algorithms (in parallel or sequentially) in dedicated time schedules. Typically one needs to trade off time for exploration (learning the performance of each method) and exploitation (executing the selected algorithm). We refer to Kotthoff (2016) for a recent survey.

For our purposes we made a selection of four methods from the literature, which contrast offline versus online learning, single versus multiple algorithm selection, and low-level versus high-level knowledge utilization. We assume that we are given a set of variable ordering heuristics (each leading to a different algorithm) and a maximum overall time limit. We explain each portfolio using

a contrived example instance in Fig. 7.2(a). Notice how one variable ordering may have exponentially longer runtime to reach the optimal value compared to other variable orderings.

7.3.1 Static Uniform Time Allocator

This multiple-algorithm selection approach proceeds in rounds; in round t, each algorithm is given 2^t seconds to solve the problem (Gagliolo and Schmidhuber 2011). We continue until the time limit is reached. As an example, see Fig. 7.2(b), where the optimal value of 10 will be reached in the round of 64s once V.O. 1 has run for 20 seconds total. So, the total runtime will be (1s+2s+4s+8s+16s+32s+5.25s)=67.25 seconds.

Runtimes	V.O. 1	V.O. 2	V.O. 3	V.O. 4
1s	5	5	5	3
2s	5	5	5	6
4s	7	6	5	6
20s	10	8	6	8
30s	10	9	8	10
60s	10	10	8	10
3000s	10	10	10	10

(a) Variable Ordering Runtimes

Uniform	V.O. 1	V.O. 2	V.O. 3	V.O. 4
1s	1/4	1/4	1/4	1/4
2s	1/4	1/4	1/4	1/4
4s	1/4	1/4	1/4	1/4
8s	1/4	1/4	1/4	1/4
16s	1/4	1/4	1/4	1/4
32s	1/4	1/4	1/4	1/4
...	...	...	...	...

(b) Uniform Time Allocator

Figure 7.2: An example of lower bounds for 4 different variable orderings at various runtimes, where the optimal value is 10, and the distribution of runtimes using a uniform time allocator.

7.3.2 Offline Predictive Models Via Classifiers

This approach uses classification models to predict the optimal algorithm to run on a given problem instance (Xu et al. 2008, Musliu and Schwengerer 2013). As input, the method requires several easily computable features of a problem instance and logic to label the best algorithm for an instance given performance data. These features and labels are computed for a training dataset, then discretized using MDL with Kononenko's criteria, and a greedy forward feature selection process. Pairwise products are computed for this subset of features as in a similar work by Xu et al. (Xu et al. 2008). Then, the same discretization and feature selection process is performed to obtain the final features and labels used to train classification models. Several classification models can be applied including Bayesian Networks (BN), Decision Trees (DT), k-Nearest Neighbor (kNN), Multilayer Perceptrons (MP), Random Forests (RF), and Support-Vector-Machines (SVM). The trained classification model is used to select one algorithm from the portfolio to solve a given test instance. For the example in Fig. 7.2(a), suppose the model takes t seconds to predict V.O. 1. The total runtime will be $t + 20$s. If alternatively the model predicts V.O. 3, then the runtime will be $t + 3000$s. This demonstrates two things: t affects the overall performance of the predictive model approach, and the predictive model choosing one single variable ordering could be detrimental.

7.3.3 Low-Knowledge Single Algorithm Selection

This is a single-algorithm selection method that runs in two phases (Beck and Freuder 2004). An exploration phase runs each algorithm for a time t, and then an exploitation phase chooses one algorithm to run for the remaining time based on the results of the exploration phase. In (Beck and Freuder 2004), three prediction rules for the exploitation phase are proposed: pcost_max (select algorithm with best lower bound), pslope_mean (maximum mean of the change in the best lower bound), and pextrap (extrapolate pcost_max with pslope_mean to find the maximum lower bound at the time limit). Ties are broken by choosing the ordering with the best mean performance at the time limit for the training data. For each prediction rule, the optimal time t to use on the testing data is found by running $t = 10, 20, \ldots, 300$ on the training instances and choosing the t that gives the maximum number of optimal lower bound results. For the example in Fig. 7.2(a), suppose the model trains for 30 seconds with each variable ordering. Using pcost_max, the model would choose the ordering with the better mean performance on the training data between V.O. 1 and V.O. 4 as these have the highest bounds, pcost_slope would choose V.O. 4 as it has the highest mean change of 0.2 per second if we start from an offset of 10 seconds, and pcost_extrap would choose V.O. 4 by calculating the highest extrapolated value of 10 + 0.2*(timeout - 30s).

7.3.4 Dynamic Online Time Allocator

This is a multi-algorithm selection method following a dynamic online schedule (Gagliolo and Schmidhuber 2011). It proceeds in rounds, such that round t has a limit of 2^t seconds. We initially assign to each algorithm a share of the runtime. After each round, the time share for each algorithm is updated based on a function of the problem instance features, the current runtime for each algorithm, and the performance of each algorithm. For our purposes, we use an updating function with three parameters: maximum lower bound (lb_bonus), maximum change in lower bound (delta_bonus), and a tie parameter (tie_bonus) that encourages reversion to the uniform time allocator. Given a time share allocation (vo_1, vo_2, ..., vo_k) at the beginning of a round, this function adds lb_bonus to the vo_i for the variable ordering i with the maximum lower bound at the end of the round. Similarly, delta_bonus adds to the maximum change in lower bound from the beginning of the round to the end of the round. In the case of any ties, the bonus is divided evenly amongst the tied variable orderings. In the case that all variables tie for both lb_bonus and delta_bonus, tie_bonus is added to all vo_i. After adding bonuses, all vo_i are re-normalized so that they sum to 1. Similar to the Low-Knowledge method of (Beck and Freuder 2004), we use the training instances to tune the parameters of the updating function to use on the test instances. For the example in Fig. 7.2(a), the table in Fig. 7.3 shows the distribution of runtimes for each round. Suppose lb_bonus=delta_bonus=tie_bonus=1. Then, no bonus is given until each variable ordering runs for 1 second, where then the lb_bonus and delta_bonus are split between V.O. 1, V.O. 2, and V.O. 3, creating the distribution for the 8s round. Then, V.O. 1 receives the lb_bonus as it passes 4s total runtime with a bound of 7 while V.O. 4 receives the delta_bonus as it passes 2s total runtime with a bound of 6. The next round would use the tie_bonus, as no variable ordering improves in the 16s round. This portfolio reaches the optimal lower bound at 39.8 seconds in the 32s round when V.O. 1 reaches 20 seconds of runtime.

Dynamic	V.O. 1	V.O. 2	V.O. 3	V.O. 4
1s	1/4	1/4	1/4	1/4
2s	1/4	1/4	1/4	1/4
4s	1/4	1/4	1/4	1/4
8s	11/36	11/36	11/36	3/36
16s	47/108	11/108	11/108	39/108
32s	155/540	119/540	119/540	147/540

Figure 7.3: The distribution of runtimes using the Dynamic Online Time Allocator.

7.4 Case Study: Graph coloring

We next apply the variable ordering portfolios for decision diagrams to graph coloring as a case study. Given a graph, the graph coloring problem is to minimize the number of colors necessary to color all vertices such that no vertices sharing an edge have the same color.

A decision diagram approach for graph coloring was proposed in (van Hoeve 2020b, 2022), using iterative refinement based on conflict resolution. The decision diagram represents the independent sets (color classes) of the graph, where each layer corresponds to a vertex of the input graph. That is, each r-t path in the decision diagram correspond to a color class defined by the vertices that take an arc with label 1 in the path. A graph coloring solution consists of a set of color classes such that each vertex belongs to one color class. To find such solution, we can define a network flow optimization model on the decision diagram, that 1) minimizes the total amount of flow out of the root node, while 2) ensuring that in each layer at least one arc with label 1 is traversed. The optimal network flow solution thus corresponds to a collection of r-t paths that 'cover' all vertices. If one of these paths contains a conflict, the decision diagram is refined accordingly, and a new network flow solution is computed. This process iterates until a conflict-free solution is found or a stopping criterion is met. The experimental results in (van Hoeve 2022) demonstrate that the performance of this approach relies strongly on the variable ordering, which makes this a relevant case study for our portfolio approach.

7.4.1 Variable Orderings

We consider the following six variable orderings, the first three of which were also studied in (van Hoeve 2022):

Lexicographic: Order the variables as they are input into the problem.

Maximum Connectivity/Degree: Add vertices one at a time, choosing the one with the maximum number of dependencies already in the ordering, and the one with the largest degree as a tie-breaker (van Hoeve 2020b).

DSATUR: Use the classic graph coloring heuristic from Brélaz (1979).

Maximal Paths: Use a maximal path decomposition to order the variables (Bergman et al. 2012a). Start by considering the variables in the order they were entered. While not all vertices are in the ordering, choose the first unchosen vertex and create a maximal path, adding vertices to the ordering as they are added to the path, then remove that maximal path from the graph.

Algorithm 6 Minimum Width Variable Ordering Algorithm

Input: Graph $G = (V, E)$
Output: Ordered list of vertices L
Definition: $\deg(v, G)$ is the degree of v in G.

$L \leftarrow \emptyset$
while V not empty **do**
  $N \leftarrow \underset{v \in V}{\operatorname{argmin}}\{\deg(v, G)\}$
  $V \leftarrow V - N$
  $E \leftarrow E - \{(i, v) : (i, v) \in E, v \in N\}$
  $L \leftarrow N{:}L$ {add N to front of L}
  $G \leftarrow (V, E)$
return L

To create the maximal path, choose an unchosen vertex that is adjacent to the most recent vertex added to the path, or if this does not exist, add an unchosen vertex adjacent to the first vertex in the path until there are no more possible vertices to add.

Maximal Cliques: Use a maximal clique decomposition to order the variables. Sort the vertices from largest degree to smallest degree. Construct a maximal clique decomposition on the graph of variable dependencies by choosing one vertex at a time, and then the first neighbor in the adjacency list that maintains a clique if one exists. Order the vertices by starting with the largest clique, and continue with cliques that share as many edges with the previous clique as possible.

Minimum Width: Apply a variable ordering with minimum *width*, that is, the maximum number of dependencies for a variable that come before that variable in the ordering (Freuder 1982). The algorithm is described in Algorithm 6.

In our evaluation, we will refer to the above orderings as 'lex', 'max_degree', 'dsatur', 'max_path', 'max_clique', and 'min_width', respectively. We note that the latter two orderings have not been applied before to decision diagram compilation, to the best of our knowledge. We give details for constructing each type of portfolio below.

7.4.2 Algorithm Portfolios

Static Uniform Time Allocator For the uniform time allocator, the order the heuristics run in each round is: min_width, max_clique, dsatur, max_degree, max_path, lex. This order was chosen based on which variable orderings solved the most instances of the Dimacs benchmark set within a 3600s time limit.

Offline Predictive Models Via Classifiers We used Culberson's random instance generator (Culberson and Luo 1996) to generate 432 graphs. We generated 4 graphs of each type in the cross product of n=(100, 250, 500, 1000), density=(0.1, 0.5, 0.9), embedded colorings of (0, 10, 20), (0, 25, 50), (0, 25, 100) and (0, 50, 100) for each n respectively, and variability=(0, 1) when the embedding does not equal 0. We use 3 graphs of each type as a training set (324 graphs), and the 4th graphs as a testing set (108 graphs). We ran each algorithm on these graphs for a maximum of 1,800 seconds. We also used a set of 137 graphs from the coloring and clique part of the well-established Dimacs

Challenge (Johnson and Trick 1996) as another, completely independent, test set. The Dimacs experiments ran with a time limit of 3,600 seconds.

For the features, we calculate 50 characteristics of each problem instance. We use a subset of the features from Musliu and Schwengerer (Musliu and Schwengerer 2013), by including only these categories: graph size features, node degree statistics, maximal clique statistics, local clustering coefficient statistics, weighted local clustering coefficient statistics, and dsatur greedy coloring statistics. Graph Size features and node degree statistics use their common definitions. Maximal clique uses a simple greedy algorithm for each node. Clustering coefficients use their classic definition (Watts and Strogatz 1998), and weighted clustering coefficients multiplies each clustering coefficient for a node by its degree. DSATUR runs the common algorithm mentioned earlier in this paper. Problem instances were labelled with a best algorithm based first on maximum lower bound, then best time to the best lower bound, and then most instances solved to optimality. To simplify parameter configuration for the classification models, we used parameters recommended in (Musliu and Schwengerer 2013). For the BN, the maximum number of parent nodes is set to 5. For the DT, the minimum number of objects per leaf was set to 3. For kNN, the size of the neighborhood is set to 5. For the RF, the number of trees was set to 15. For the MP and SVM, and other remaining parameters, we used the default settings from the Weka system (Bouckaert et al. 2016).

Low-Knowledge Single Algorithm Selection We order the variables as we did in the Static Uniform Time Allocator, and we use the training set from the Offline Predictive Models Via Classifiers to find the best parameter t. When determining the mean change for pslope_mean, we consider the interval from 10 seconds to the end of the training phase, as all variable orderings start with a lower bound of 1.

Dynamic Online Time Allocator We order the variables as we did in the Static Uniform Time Allocator, and we use the training set from the Offline Predictive Models Via Classifiers to find the best set of bonus parameters.

7.5 Experimental Evaluation

All variable orderings and iterative refinement algorithms are written in C++. The data evaluation scripts are written in Python, using a wrapper around the Weka data mining library version 1.0.6 for the machine learning models used (Bouckaert et al. 2016). Following previous studies, we assume an "ideal" machine with no task switching overhead (Gagliolo and Schmidhuber 2011). Therefore, our experiments were run for each single variable ordering, and this data was compiled to simulate each portfolio method. All experiments were run on an Intel Xeon 2.33GHz CPU with Ubuntu 18.04. We will evaluate each algorithm (i.e., the individual variable orderings and portfolios) in terms of their performance: how many instances can be solved within a given time limit? We first consider the performance of the individual variables orderings, and then assess each of the four portfolio approaches from Section 7.3.

7.5.1 Performance of individual variable orderings

Before running our portfolio methods, we confirm that none of the individual orderings always dominates the others, and that each ordering can be the best for at least one instance. We show that this is the case for both the Test Culberson instances and Dimacs instances through Figure 7.4(a), which plots the frequency that a variable ordering achieved the best bound amongst all variable

orderings within three time ranges of when the quickest variable ordering achieved that bound. For the 108 Test Culberson instances, any one variable ordering achieves the best lower bound within 60 seconds of the quickest variable ordering to achieve this bound for less than 80 instances. Similarly, this number is 100 of the 137 Dimacs instances.

A more detailed comparison of the individual variable orderings is given in Fig. 7.5 by presenting their performance plots, i.e., the number of instances solved by a given time limit. We show these plots separately for the test Culberson instances (a) and the Dimacs instances (b). The best performing individual variable orderings for the Culberson instances are dsatur and max_degree (both solve 46 instances). While max_degree solves 46 instances in 1200 seconds, dsatur solves 46 instances in 1740 seconds. For the Dimacs instances the min_width ordering performs best (solving 54 instances). The min_width ordering also performs best overall, solving 98 instances in total compared to 96 for the runner up max_degree.

7.5.2 Experiment 1: Static Uniform Time Allocator

We included this method as a baseline comparison. Despite its simplicity, the uniform time-sharing portfolio solves 55 Dimacs instances optimally, and solves more Dimacs instances in faster times than all of the variable orderings individually, as can be seen in Fig. 7.5(b). This method also works well, but not as well, on the Test Culberson instances, as presented in Fig. 7.5(a). In both cases, there is at least one instance that a hypothetical 'oracle' portfolio, which selects the best variable ordering for each instance can solve, but the uniform portfolio cannot.

7.5.3 Experiment 2: Offline Predictive Models Via Classifiers

The predictive model used a greedy forward feature selection which chose 28 features (4 basic features and 24 product features) ranging over all of the feature categories (the same features were used for all models). The Multilayer Perceptrons model (MP) took 10 minutes to train, while the other models needed less than one minute to train, so we chose to not include results for MP. All of the testing took less than a second. Among all instances, the median time taken to compute all features for an instance is 1 second, the 75th percentile is 19 seconds, and the maximum is 2562 seconds. Most classifiers showed similar performance as seen in Fig. 7.6. Random Forests (RF) performed best for Culberson instances solving 45 instances, and Bayesian Network (BN) for Dimacs instances solving 55 instances. The results highlight the fact that the models are trained on Culberson data, so the Culberson test results simulate a user having access to results from a similar problem set, while the Dimacs results simulate a user lacking similar training data.

7.5.4 Experiment 3: Low-Knowledge Single Algorithm Selection

Recall that for six orderings and a time limit T, the training phase for this method takes $6 * t$ seconds, while the the final selected algorithm runs for a total of $t + (T - 6 * t)$ seconds. As stated before, we use $T = (3600, 1800)$ for the Dimacs and Culberson Test sets respectively. Based on the results of the training data, we set $t = (30, 10, 10)$ for pcost_max, pslope_mean, and pextrap respectively. We present the performance for the three possible settings for this type of portfolio in Fig. 7.7. We see that pcost_slope performs best for Culberson instances solving 46 instances, while pcost_max performs best for Dimacs instances, solving 56 instances. The choice of 10 seconds for pslope_mean will always select the top tie-breaker option (which was dsatur), because there is no slope to calculate with only one bin, so we use pcost_max for both types of instances in our final

Table 7.1: Number of instances solved to optimality within three time limits for the Culberson and Dimacs instance types. We compare the best single variable ordering, each of the portfolio approaches, the oracle, and the method by Held et al. (Held et al. 2012).

	Culberson			Dimacs		
	100s	750s	1,800s	100s	1,000s	3,600s
Single Variable Ordering (max_degree/min_width)	36	44	46	51	52	54
Static Uniform Time Allocator	35	40	44	51	54	55
Offline Predictive Model (RF/BN)	35	43	45	49	54	55
Low-Knowledge Single Algorithm (PCOST)	32	42	45	40	54	56
Dynamic Online Time Allocator	37	45	46	53	53	55
Oracle	39	44	46	54	54	57
Held et al.	44	50	52	53	56	60

comparisons. The results show that while this type of portfolio may take a while to train, its late performance looks strong for both types of instances.

7.5.5 Experiment 4: Dynamic Online Time Allocator

We ran this method on the training data using values of (0, 2, 4, 6) for each possible bonus value. Based on those results, for the testing sets we used lb_bonus = 6, delta_bonus = 6, and tie_bonus = 6. These large yet equal bonuses made it quick to either converge to an optimal allocation share or revert back to the uniform distribution. The overall comparison in the next section includes these results.

7.5.6 Overall Comparison

Lastly, we compare the performance of the best settings for each type of portfolio against the oracle, the best performing individual ordering (max_degree, resp. min_width), and the state-of-the-art graph coloring solver by Held et al. (Held et al. 2012). The latter solver is based on integer linear programming, and implements a branch-and-price algorithm.[1] The performance plot for each method is given in Fig. 7.8 for the Culberson instances (a) and Dimacs instances (b). In addition, Table 7.1 presents the number of instances solved to optimality within three different time limits for each algorithm and each instance type.

Table 7.1 shows that the best performing portfolios at the time limit are PCOST and dynamic_online for the Culberson instances (solving 46 instances), and PCOST again for the Dimacs instances (solving 56 instances). Fig. 7.8 furthermore shows that also in terms of overall performance (across varying time limits and for both instance types), both the low-knowledge PCOST and dynamic online portfolios perform well. However, one may favor the dynamic online portfolio because the low-knowledge method is often slower to reach optimality due to its training phase. Notice that for the Culberson instances, the dynamic_online portfolio solves the same number of instances as the single variable ordering max_degree. More granularity shows that the dynamic_online portfolio

[1]The code has been downloaded from `https://github.com/heldstephan/exactcolors`.

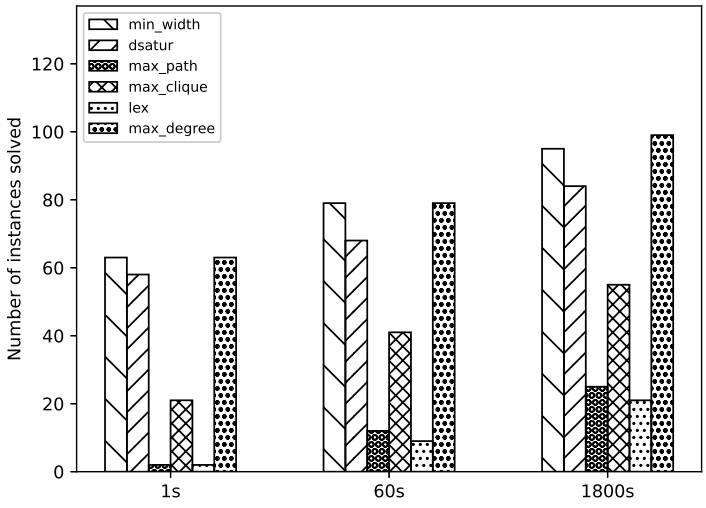
solves 46 instances in 1,080 seconds while the max_degree ordering takes 1,200 seconds. This indicates that for a set of similar graph instances, one variable ordering might work just as well as a portfolio, but when the set of instances is more diverse like in the Dimacs set, the portfolio can become more helpful.

The predictive method shows slightly stronger relative performance to the other portfolio methods on the Culberson instances than for the Dimacs instances, especially looking at 100s. This is likely because the training set consists of Culberson instances. Also notice from comparing the performance of the dynamic online portfolio and the oracle at 750s for Culberson instances that portfolios using more than one ordering can even outperform the oracle when one variable ordering finds the best lower bound and another finds the matching upper bound.

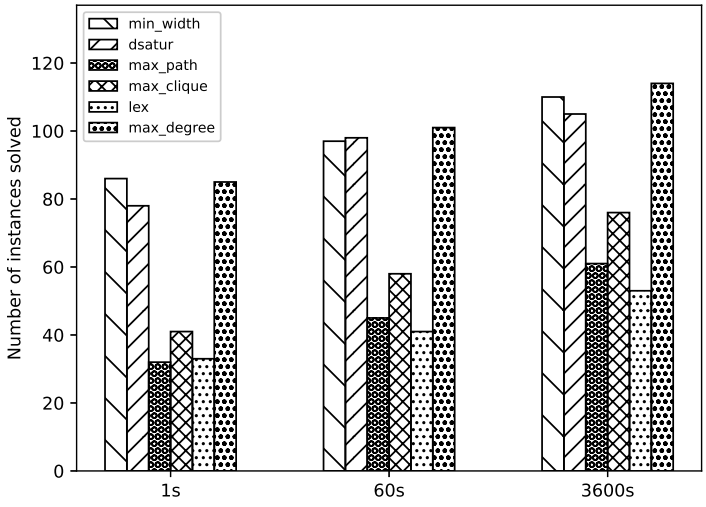
In a comparison to the state of the art, the dynamic online portfolio improves the performance of the decision diagram approach to be competitive with Held et al. for the Dimacs instances. However, overall Held et al. solve more instances within the time limit. For the Culberson instances, the approach by Held et al. clearly outperforms the best portfolio approach.

7.6 Conclusion

We presented a portfolio approach to selecting the best variable ordering for relaxed decision diagrams in the context of combinatorial optimization. We considered four approaches: uniform time allocation, predictive modeling, a low-knowledge selection procedure, and a dynamic online time allocator. We compared the performance of these methods on the graph coloring problem, and find that even the simplest portfolio (uniform time allocation) already outperforms all individual orderings for the Dimacs benchmark set of instances. The dynamic online time allocator showed the best overall performance. As it can combine lower and upper bounds from different orderings, it is even able to outperform an oracle that selects the best single ordering for each instance.

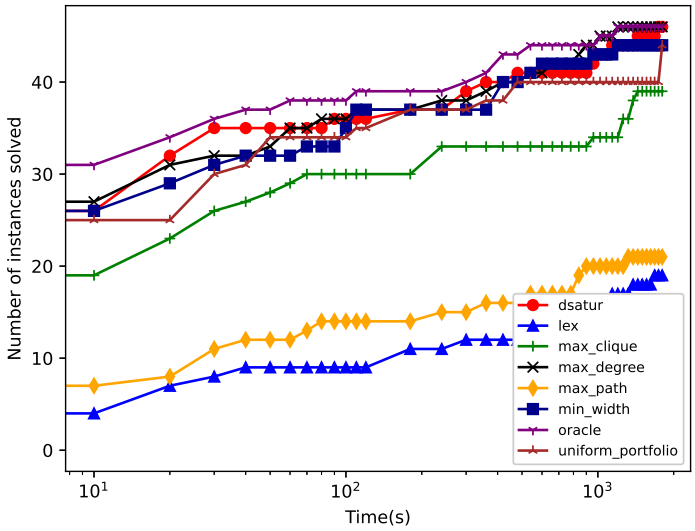


(a) Test Culberson instances

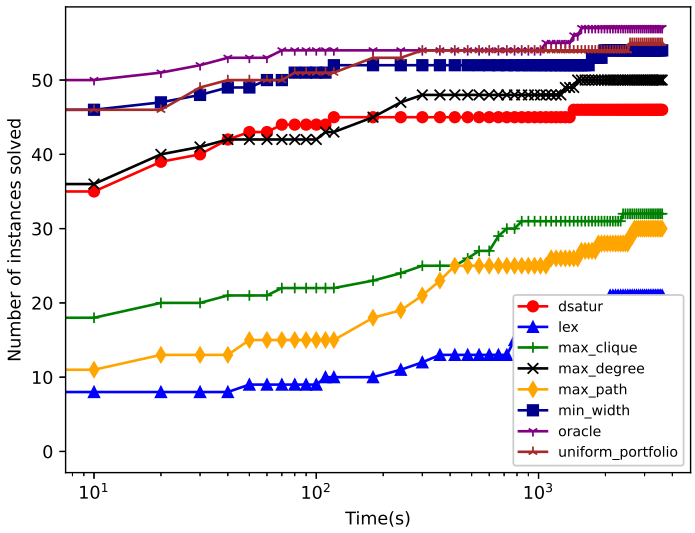


(b) Dimacs instances

Figure 7.4: The frequency that a variable ordering yields the best lower bound within a time range of (1s, 60s, 1800/3600s) from the fastest time of any ordering, for the Test Culberson instances (a) and the Dimacs instances (b).

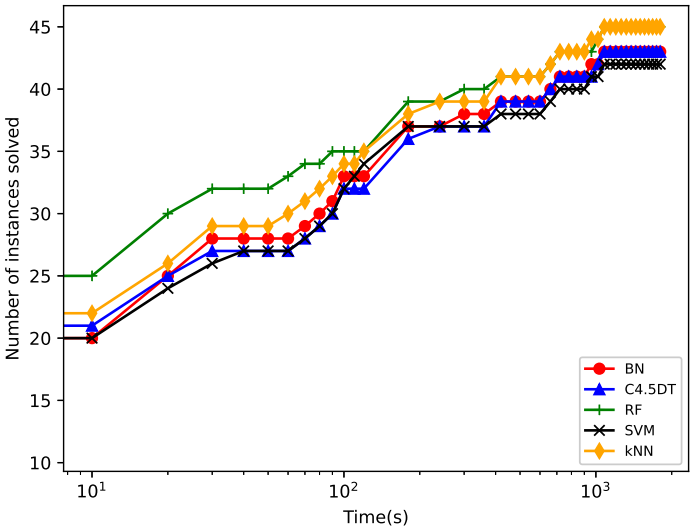


(a) Test Culberson Instances

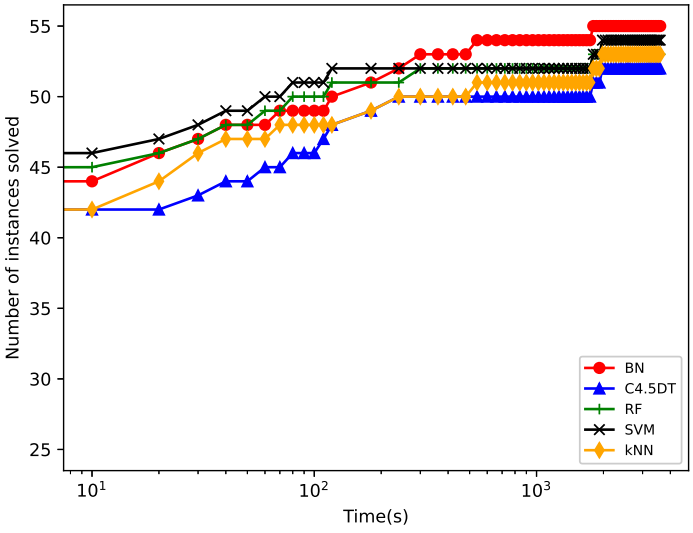


(b) Dimacs Instances

Figure 7.5: The number of instances solved to optimality within t seconds for each variable ordering, the oracle, and the uniform time-sharing portfolio. The time is in log-scale.

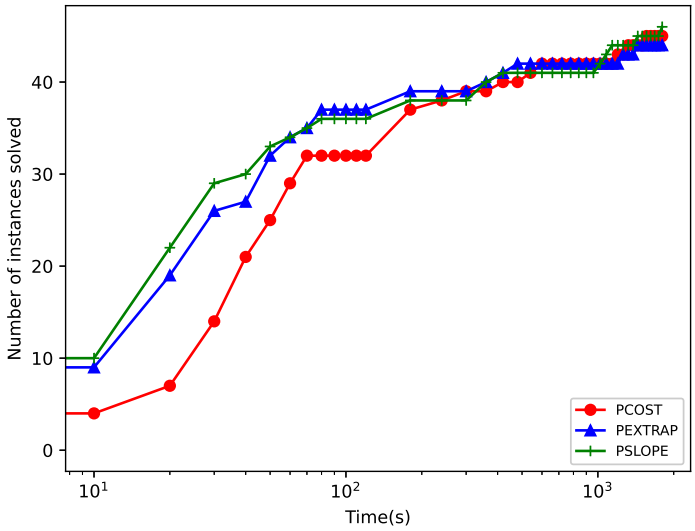


(a) Test Culberson Instances

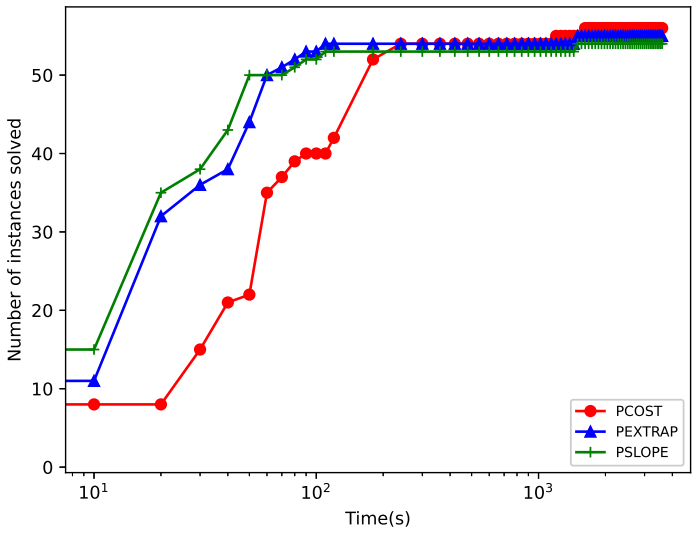


(b) Dimacs Instances

Figure 7.6: The number of instances solved to optimality within t seconds for each type of classifier used in the predictive method and the oracle. The time is in log-scale.

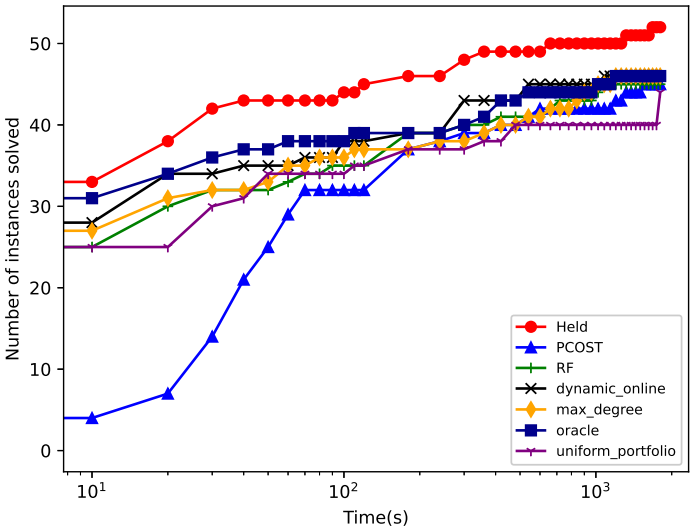


(a) Test Culberson Instances

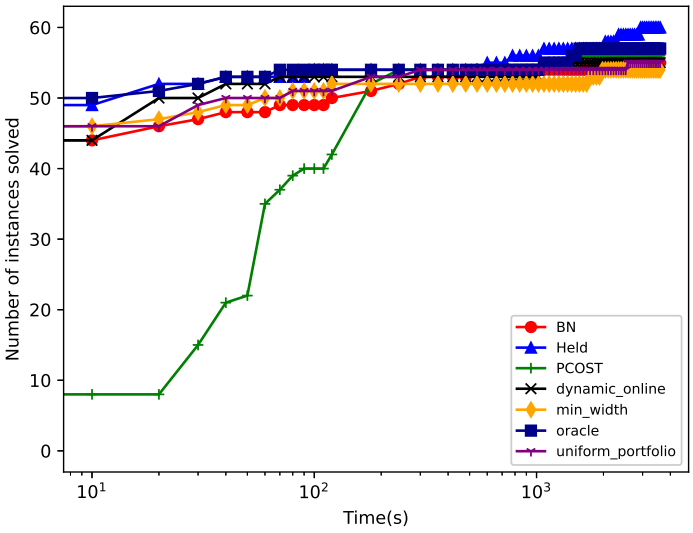


(b) Dimacs Instances

Figure 7.7: The number of instances solved to optimality within t seconds for each function that can be used in the low knowledge portfolio and the oracle. The time is in log-scale.



(a) Test Culberson instances



(b) Dimacs instances

Figure 7.8: The number of instances solved to optimality within t seconds for the best performing individual variable ordering (max_degree/min_width), the best setting for each portfolio method, the oracle, and the state-of-the-art code by Held et al. (Held et al. 2012), for the Test Culberson instances (a) and the Dimacs instances (b).

Chapter 8

Conclusion

This dissertation does not show that column elimination always outperforms existing approaches. Rather, it develops and describes the novel framework, studies its computational performance, and shows some instances where it performs well.

The current work on column elimination has so far revealed two types of problems for which it performs well. First, problems where relaxations are not often considered, such as the graph coloring problem where column elimination relaxes the notion of an independent set. Second, problems where the linear programming relaxation of the arc flow formulation provides a small optimality gap, which enables the combined power of a Lagrangian method, a refinement algorithm, and variable fixing.

Column elimination has features that distinguish it from the existing iterative refinement approaches that solve arc flow formulations, which were mentioned in Section 2. Each method starts with a relaxed model such as a state-space relaxation, solves the relaxed model, and uses the solution to refine the relaxed model. The following are comments about these components in the context of solving arc flow formulations.

Consider the initial model relaxation. There is a tradeoff between the size of the model and the strength of the relaxation. State-space relaxations and discretizations are the most common ways to relax a dynamic program, which is an equivalent way of relaxing an arc flow formulation. These relaxations often maintain the true cost of each sequence and only relax the set of feasible sequences. Aggregations and merging techniques from decision diagram approaches can offer more flexibility to the relaxations. The column elimination framework allows this flexibility, although the works presented in this thesis only use state-space relaxations for the initial relaxation.

Consider the method for solving the relaxed model at each iteration. The following are three important considerations: the efficiency of the method, the ability to add cutting planes, and when refinement can occur. Integer programming solvers and linear programming solvers are simple to use and can automatically identify and add cutting planes. However, they can be inefficient for large instances and require waiting for the optimal solution to make a refinement. Branch-and-price is often efficient, easily allows adding robust cutting planes, and permits refinements based on solutions to the pricing problem. However, it can have issues with stability and convergence, struggle with non-robust cuts, and often uses coarse refinements to maintain an efficiently solvable pricing problem. Lagrangian methods can be efficient as long as the Lagrangian subproblem can be solved efficiently. They give feasible dual solutions at each iteration which provides dual bounds that can be used in variable fixing. Refinement can occur at each iteration based on the solution to

the Lagrangian subproblem. However, they can have convergence issues, and it is difficult to add cuts to Lagrangian methods. So far column elimination relies on integer programming solvers and Lagrangian methods.

Consider the refinement algorithm. Refinement algorithms vary a lot. They can be based on the specific infeasibilities that the relaxed model allows. For example, a specialized refinement algorithm can remove infeasible solutions that are caused by repeating an element in a sequence. Similarly, a specialized refinement algorithm can leverage particular types of discretizations of values. On the other hand, some refinement methods can generically remove an infeasible sequence. Based on the method of solving the relaxed model, some refinement algorithms very coarsely update a state-space relaxation while others make minor changes to the state-transition graph. The mechanics for most refinement algorithms are based on disaggregating nodes or introducing nodes and updating arcs to fix an infeasibility. Column elimination uses a refinement algorithm that is generically defined for dynamic programs and makes minor changes to the state-transition graph.

There are several promising directions for future work. First, column elimination can solve more types of problems. For example, the current setup requires a homogeneous set of sequences, but this can be extended to sequences from different sets. Also, current work has only considered vehicle routing problems and vertex coloring problems. Second, the method for solving the relaxation can be improved. The Lagrangian method can be improved in several ways including warm-starting the successive shortest paths algorithm at each iteration, implementing a bundle method instead of subgradient descent, and a better cut-and-refine approach to include cutting planes. Column generation can also be incorporated as a method to solve the relaxation. Third, the relaxations used by column elimination can be improved. The initial relaxation can be created by merging methods from decision diagrams instead of using state-space relaxations. Good initial relaxations could be learned with machine learning. Finally, instead of only refining conflicts, a method can be used to merge nodes in the state-transition graph.

Appendices

Appendix A

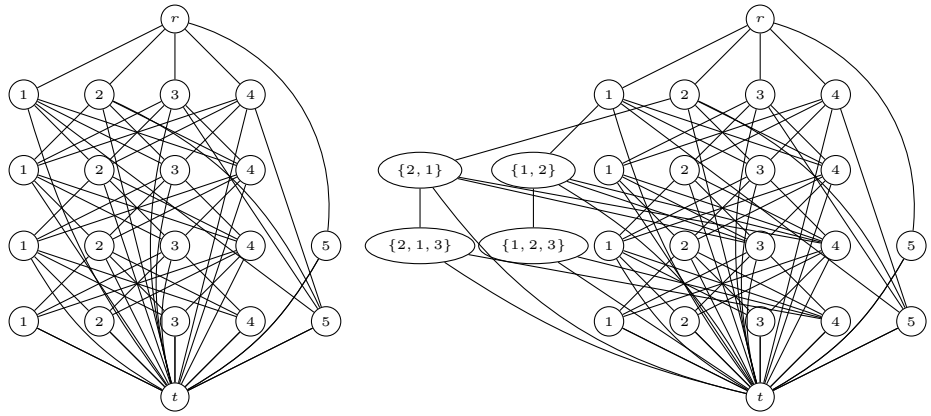
Dynamic Program Relaxation Example

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. (1981b). 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 9, using data for an example CVRP instance in Figure 3.2. The costs are not relaxed in either dynamic program, so we only argue about the solution sets complying with the definitions.

Proposition 9. The dynamic program relaxation represented by the state-transition graph in Figure A.1(b) is not a state-space relaxation.

Proof. 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. □



a. State-space relaxation.

b. Dynamic program relaxation.

Figure A.1: 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.

Appendix B

Application Models

B.1 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 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 $(\text{NG}, w, v, \tau, c)$ with the initial state being $r_1 = (\emptyset, 0, 0, 0, 0)$. The set of states S' is implicitly in defining a transition function h'. Given a state $s = (\text{NG}, w, v, \tau, c)$ and label $u \in U$ such that $u \neq 0$, $u \notin \text{NG}$, $w + q_u \leq Q$, $\tau + \ell_{v,u} \leq l_u$, and $c < U$, let the transition function be $h'(s, u) = ((\text{NG} \cup \{u\}) \cap N_u, (w + q_u) * (1 - \kappa), u, \max(\lfloor \frac{\tau + \ell_{v,u}}{\Delta} \rfloor * \Delta, e_v), c + 1)$. 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 $s = (\text{NG})$ 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 $s = (\text{NG})$ and decision i such that $i \notin \text{NG}$ is $h(s, i) = (\text{NG} \cup N_i \cup \{1, \ldots, i\})$, 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 $s = (\text{NG})$ 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 $h'(s, u) = (\{1, \ldots, u\} \cup N_u)$. 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 PDPTW / 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 model the SOP in this way, with the additional constraint that a solution has one sequence.

Appendix C

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 9 generalizes the arc fixing theorem from Karahalios and van Hoeve (2023b), 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 9. 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 $v(\nu) + sp_{v_1}^{\downarrow} + sp_{v_2}^{\uparrow} + rc(a) > UB$, then arc a can be fixed to have flow 0 in F.

Proof. 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.

$$(IP) \quad \min \sum_{x \in \mathcal{X}} z_x f(x) \tag{C.1}$$

$$\text{s.t.} \sum_{x \in \mathcal{X}} z_x \sum_{a \in \mathcal{A}_x} g_j(a) \circ_j b_j \qquad \forall\, j \in \{1, \dots, m\} \tag{C.2}$$

$$z \in \mathbb{Z}_+^{|\mathcal{X}|} \tag{C.3}$$

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 $p = \{a_1, \dots, a_l\}$ 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 $rc'(x') =$

$sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(a)$. 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, $v(\nu) - \sum_{j=1}^{m} \nu_j \sum_{a \in \mathcal{A}_{x''}} g_j(a) + f(x'') = v(\nu) + rc(x'') \geq v(\nu) + rc(x') \geq v(\nu) + sp^{\downarrow}_{v_1} + sp^{\uparrow}_{v_2} + rc(a) > UB$. □

Appendix D

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 D.1(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 D.1(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

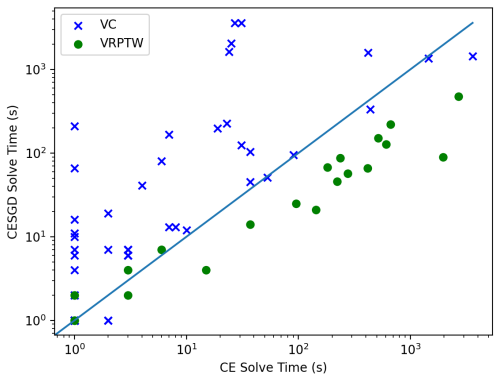
Figure D.1: Sequences of dual solutions obtained by running a column elimination and column elimination with subgradient descent to solve the VRPTW instance C1_2_5. The dual solutions are plotted in three dimensions using the Python package 'sklearn'.

Appendix E

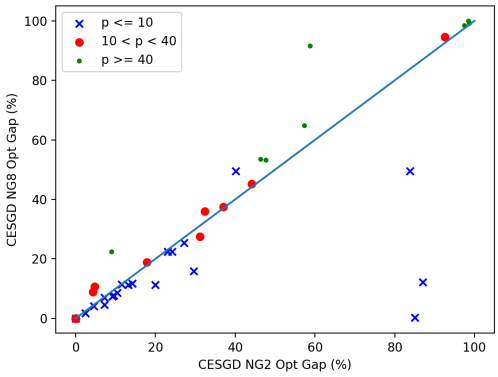
Evaluating Column Elimination with Subgradient Descent

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 E.1(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 E.1: 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 F

Sensitivity to Initial Relaxations

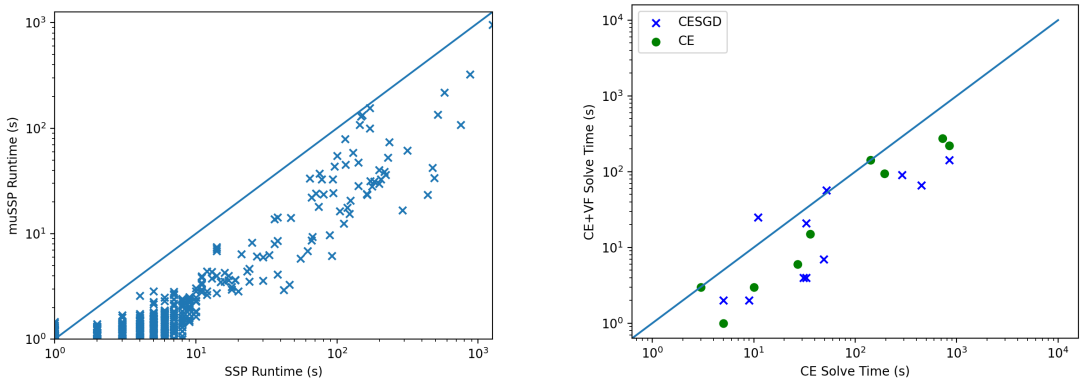
We evaluate the sensitivity of column elimination to the initial relaxation. To do this, we use column elimination with subgradient descent and two different initial relaxations to solve the TSPLIB SOP instances (Ascheuer et al. 2000). The two initial relaxations are ng-route with $\rho = 2$ and $\rho = 8$. 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 E.1(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.

Appendix G

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 G.1(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.



a. Impact of muSSP

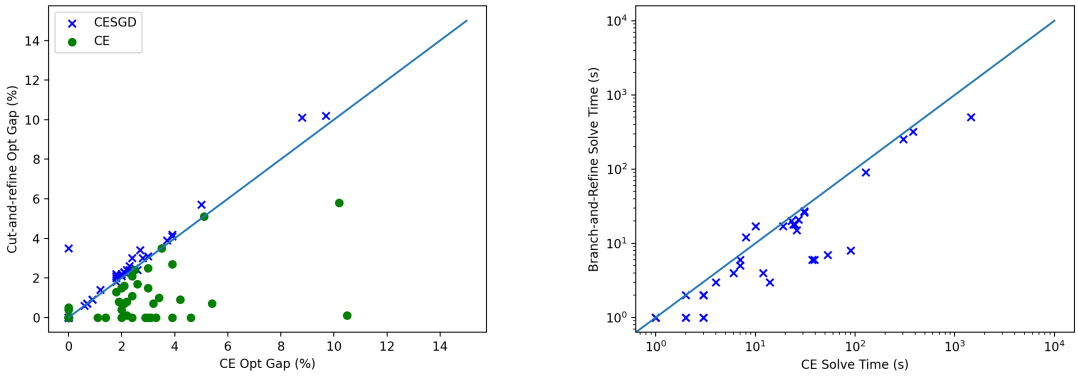
b. Impact of Variable Fixing

Figure G.1: 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.

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 G.1(a). Variable fixing allows column elimination with subgradient descent to solve one additional instance and solved instances on 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.



a. Cut-and-refine

b. Branch-and-refine

Figure H.1: 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.

Appendix I

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 4.3. 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/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 (2023b).

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 H.1(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.

Appendix J

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 vertex coloring instances using branch-and-cut and column elimination Johnson and Trick (1996).

We plot the runtimes of column elimination and branch-and-refine on instances that both methods solve in Figure H.1(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.

Appendix K

VRPTW Results

Table K.1: 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	42389.3	1	290	742	0	3600
C1_10_10	39816.8	-	-	3600	37913.0	1	24	3814	0	3600
C1_10_2	41337.8	41068.76	1	3600	39383.6	1	48	3954	0	3600
C1_10_3	40064.4	-	-	3600	38051.3	1	12	2637	0	3600
C1_10_5	42434.8	42434.8	1	1227	41924.9	1	176	4239	0	3600
C1_10_6	42437	42437.0	1	1670	41184.4	1	120	6296	0	3600
C1_10_7	42420.4	42305.87	1	3600	40791.3	1	125	5321	0	3600
C1_10_8	41652.1	41062.0	1	3600	39042.5	1	76	6491	0	3600
C1_10_9	40288.4	39508.04	1	3600	38150.4	1	44	4953	0	3600
C1_2_1	2698.6	2698.6	1	11	2698.6	4	1	3	0	11
C1_2_10	2624.7	2624.7	1	218	2522.82	3	861	8265	0	3600
C1_2_2	2694.3	2694.3	1	29	2694.3	12	121	1099	41	821
C1_2_3	2675.8	2675.8	3	338	2614.92	1	680	5772	0	3600
C1_2_4	2625.6	2625.6	1	457	2516.12	1	414	6904	0	3600
C1_2_5	2694.9	2694.9	1	16	2694.9	6	1	12	1	84
C1_2_6	2694.9	2694.9	1	20	2694.9	6	12	79	2	129
C1_2_7	2694.9	2694.9	1	18	2694.9	7	11	85	4	173
C1_2_8	2684	2684.0	1	26	2680.18	24	124	2178	48	3600
C1_2_9	2639.6	2639.6	1	65	2578.09	6	1150	9021	0	3600
C1_4_1	7138.8	7138.8	1	99	7138.8	4	12	23	1	50
C1_4_10	6825.4	6820.19	1	3600	6608.93	1	226	7098	0	3600
C1_4_2	7113.3	7113.3	7	587	7046.75	1	349	3188	0	3600
C1_4_3	6929.9	6929.9	1	991	6769.86	1	158	4852	0	3600
C1_4_4	6777.7	6769.16	1	3600	6578.63	1	79	4988	0	3600

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
C1_4_5	7138.8	7138.8	1	134	7138.8	3	33	235	2	258
C1_4_6	7140.1	7140.1	1	204	7140.1	4	73	1096	3	547
C1_4_7	7136.2	7136.2	1	185	7116.39	45	100	2089	25	3600
C1_4_8	7083	7083.0	5	1128	6917.48	1	461	7893	0	3600
C1_4_9	6927.8	6927.8	1	1547	6702.82	1	306	8549	0	3600
C1_6_1	14076.6	14076.6	1	292	14076.6	4	43	102	1	224
C1_6_10	13617.5	13520.25	1	3600	13132.2	1	84	6131	0	3600
C1_6_2	13948.3	13948.3	15	1616	13725.8	1	157	3191	0	3600
C1_6_3	13757	13702.24	1	3600	13285.1	1	58	4470	0	3600
C1_6_4	13538.6	13347.86	1	3600	13026.3	1	24	2556	0	3600
C1_6_5	14066.8	14066.8	1	393	14066.8	3	139	1038	0	1450
C1_6_6	14070.9	14070.9	1	531	14007.8	1	318	3537	0	3600
C1_6_7	14066.8	14066.8	1	476	13999.1	1	314	3249	0	3600
C1_6_8	13991.2	13967.03	3	3600	13598.3	1	220	6853	0	3600
C1_6_9	13664.5	13649.77	1	3600	13225.7	1	136	7265	0	3600
C1_8_1	25156.9	25156.9	1	761	25156.9	3	89	311	0	674
C1_8_10	24026.7	23640.28	1	3600	22962.9	1	49	4641	0	3600
C1_8_2	24974.1	24910.3	1	3600	24094.4	1	77	4205	0	3600
C1_8_3	24156.1	23865.67	1	3600	23194.8	1	26	3668	0	3600
C1_8_4	23797.3	-	-	3600	22620.8	1	9	1627	0	3600
C1_8_5	25138.6	25138.6	1	737	25071.0	1	262	2692	0	3600
C1_8_6	25133.3	25133.3	1	1056	24841.7	1	183	4648	0	3600
C1_8_7	25127.3	25127.3	7	1140	24747.7	1	180	4032	0	3600
C1_8_8	24809.7	24688.04	1	3600	23743.9	1	126	7454	0	3600
C1_8_9	24200.4	23972.45	1	3600	23166.8	1	77	6173	0	3600
C2_10_1	16841.1	-	-	3600	16727.6	1	142	1476	0	3600
C2_10_10	15728.6	-	-	3600	12902.4	1	37	1010	0	3600

Table K.1: Continued.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
C2_10_2	16462.6	-	-	3600	14676.5	1	47	928	0	3600
C2_10_5	16521.3	-	-	3600	15453.4	1	109	3093	0	3600
C2_10_6	16290.7	-	-	3600	14834.9	1	78	2036	0	3600
C2_10_7	16378.4	-	-	3600	14627.6	1	59	1507	0	3600
C2_10_8	16029.1	-	-	3600	14079.1	1	55	1491	0	3600
C2_10_9	16075.4	-	-	3600	13297.9	1	37	1038	0	3600
C2_2_1	1922.1	1922.1	1	228	1922.1	10	46	19	0	131
C2_2_10	1791.2	1791.2	1	631	1681.9	1	596	4145	0	3600
C2_2_2	1851.4	1851.4	1	387	1819.17	1	352	2095	0	3600
C2_2_3	1763.4	1753.63	3	3600	1668.03	1	237	1907	0	3600
C2_2_4	1695	1666.49	3	3600	1522.69	1	138	1096	0	3600
C2_2_5	1869.6	1869.6	1	276	1847.4	6	317	2291	16	3600
C2_2_6	1844.8	1844.8	1	249	1787.48	2	901	6230	0	3600
C2_2_7	1842.2	1842.2	1	170	1790.72	3	772	5354	0	3600
C2_2_8	1813.7	1813.7	1	222	1732.38	1	723	4992	0	3600
C2_2_9	1815	1815.0	1	511	1728.32	1	586	4173	0	3600
C2_4_1	4100.3	4100.3	1	852	4085.95	29	150	992	0	3600
C2_4_10	3665.1	3647.88	1	3600	3397.41	1	175	2128	0	3600
C2_4_2	3914.1	3900.22	1	3600	3815.14	1	152	2153	0	3600
C2_4_3	3755.2	3723.96	1	3600	3348.42	1	79	1021	0	3600
C2_4_4	3523.7	3486.12	1	3600	2725.34	1	51	474	0	3600
C2_4_5	3923.2	3923.2	1	971	3831.85	1	376	5034	0	3600
C2_4_6	3860.1	3860.1	1	2466	3696.11	1	291	3892	0	3600
C2_4_7	3870.9	3870.9	1	1483	3692.25	1	253	3391	0	3600
C2_4_8	3773.7	3770.24	1	3600	3553.38	1	232	3090	0	3600
C2_4_9	3842.1	3806.45	1	3600	3568.74	1	210	2714	0	3600
C2_6_1	7752.2	7719.46	1	3600	7688.34	1	391	1671	0	3600

Table K.1: Continued.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
C2_6_10	7123.9	6340.81	1	3600	**6437.63**	1	94	1733	0	3600
C2_6_2	7471.5	7075.15	1	3600	**7177.06**	1	94	1546	0	3600
C2_6_3	7215	4670.06	1	3600	**5953.32**	1	41	593	0	3600
C2_6_5	7553.8	7540.44	1	3600	7241.6	1	231	4427	0	3600
C2_6_6	7449.8	7400.61	1	3600	6976.78	1	168	3227	0	3600
C2_6_7	7491.3	6294.69	1	3600	**6966.47**	1	151	2871	0	3600
C2_6_8	7303.7	7223.09	1	3600	6753.56	1	140	2559	0	3600
C2_6_9	7303.2	5754.15	1	3600	**6741.86**	1	104	1834	0	3600
C2_8_1	11631.9	-	-	3600	11551.8	1	196	1177	0	3600
C2_8_10	10946	-	-	3600	9589.46	1	62	1133	0	3600
C2_8_2	11394.5	-	-	3600	10571.2	1	66	1403	0	3600
C2_8_3	11138.1	-	-	3600	7521.76	1	23	438	0	3600
C2_8_5	11395.6	-	-	3600	10829.3	1	154	3589	0	3600
C2_8_6	11316.3	-	-	3600	10462.4	1	104	2330	0	3600
C2_8_7	11332.9	-	-	3600	10403.4	1	88	1968	0	3600
C2_8_8	11133.9	-	-	3600	10059.0	1	80	1700	0	3600
C2_8_9	11140.4	-	-	3600	9941.42	1	65	1332	0	3600
R1_10_1	53046.5	52756.54	1	3600	50054.4	1	30	38	0	3600
R1_10_10	47364.6	46676.18	1	3600	-	-	-	-	-	3600
R1_10_5	50406.7	49928.13	1	3600	46544.8	1	13	118	0	3600
R1_10_9	49162.8	48632.73	1	3600	-	-	-	-	-	3600
R1_2_1	4667.2	4667.2	1	16	4645.79	5	806	189	0	3600
R1_2_10	3293.1	3285.31	15	3600	3112.62	1	53	784	0	3600
R1_2_2	3919.9	3919.9	1	45	3548.3	1	57	593	0	3600
R1_2_3	3373.9	3358.72	5	3600	2777.54	1	15	108	0	3600
R1_2_4	3047.6	3039.51	3	3600	-	-	-	-	-	3600
R1_2_5	4053.2	4053.2	3	399	3975.13	1	301	1308	0	3600

Table K.1: Continued.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
R1_2_6	3559.1	3552.78	13	3600	3261.57	1	50	678	0	3600
R1_2_7	3141.9	3141.9	1	750	2738.89	1	15	120	0	3600
R1_2_8	2938.4	2938.4	5	2030	-	-	-	-	-	3600
R1_2_9	3734.7	3734.7	3	731	3607.4	1	165	1453	0	3600
R1_4_1	10305.8	10305.8	3	338	10134.3	1	220	137	0	3600
R1_4_10	8077.8	8028.88	1	3600	7230.95	1	15	326	0	3600
R1_4_2	8873.3	8843.47	7	3600	7162.98	1	13	112	0	3600
R1_4_3	7784.3	7698.59	9	3600	-	-	-	-	-	3600
R1_4_4	7266.2	7200.12	3	3600	-	-	-	-	-	3600
R1_4_5	9184.6	9153.48	11	3600	8911.2	1	89	628	0	3600
R1_4_6	8340.4	8321.6	5	3600	-	-	-	-	-	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	8120.79	1	43	871	0	3600
R1_6_1	21274.2	21231.91	11	3600	20489.4	1	92	246	0	3600
R1_6_10	17583.7	17344.32	3	3600	-	-	-	-	-	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	18477.3	1	40	399	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	16773.9	1	17	349	0	3600
R1_8_1	36345	36225.6	5	3600	34190.2	1	49	139	0	3600
R1_8_10	30918.4	30551.1	1	3600	-	-	-	-	-	3600
R1_8_2	32277.6	31948.36	1	3600	-	-	-	-	-	3600
R1_8_5	33494.2	33279.06	1	3600	31174.3	1	22	306	0	3600

Table K.1: Continued.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
R1_8_6	30872.4	30460.46	1	3600	-	-	-	-	-	3600
R1_8_9	32257.3	31928.06	1	3600	-	-	-	-	-	3600
R2_2_1	3468	3468.0	1	142	3147.13	1	179	864	0	3600
R2_2_10	2549.4	2549.4	3	1439	1167.03	1	34	46	0	3600
R2_2_2	3008.2	3008.2	1	531	803.416	1	18	0	0	3600
R2_2_3	2537.5	2537.5	1	2283	-	-	-	-	-	3600
R2_2_4	1928.5	1925.02	3	3600	-	-	-	-	-	3600
R2_2_5	3061.1	3061.1	1	1125	1912.5	1	71	201	0	3600
R2_2_6	2675.4	2675.4	3	2384	803.416	1	18	0	0	3600
R2_2_7	2304.7	2298.61	1	3600	-	-	-	-	-	3600
R2_2_8	1842.4	1819.76	1	3600	-	-	-	-	-	3600
R2_2_9	2843.3	2843.3	1	782	1547.69	1	56	104	0	3600
R2_4_1	7520.7	7520.7	1	2828	4927.05	1	59	160	0	3600
R2_4_10	5645.9	5543.96	1	3600	-	-	-	-	-	3600
R2_4_2	6482.8	6374.27	1	3600	-	-	-	-	-	3600
R2_4_5	6567.9	6527.42	1	3600	3070.78	1	30	13	0	3600
R2_4_6	5813.5	5643.47	1	3600	-	-	-	-	-	3600
R2_4_9	6067.8	6027.42	1	3600	-	-	-	-	-	3600
R2_6_1	15145.3	-	-	3600	7667.62	1	30	10	0	3600
R2_8_1	24969.8	-	-	3600	5358.46	1	5	0	0	3600
RC1_10_1	45790.8	45302.6	1	3600	41546.0	1	16	749	0	3600
RC1_10_5	45028.1	44404.77	1	3600	-	-	-	-	-	3600
RC1_10_6	44903.6	44284.41	1	3600	-	-	-	-	-	3600
RC1_10_7	44417.1	43820.04	1	3600	-	-	-	-	-	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	3434.03	1	422	2113	0	3600

Table K.1: Continued.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
RC1_2_10	2990.5	2969.39	5	3600	2739.26	1	42	964	0	3600
RC1_2_2	3221.6	3213.54	9	3600	2976.47	1	66	973	0	3600
RC1_2_3	3001.4	2984.21	3	3600	2598.55	1	24	346	0	3600
RC1_2_4	2845.2	2833.72	3	3600	-	-	-	-	-	3600
RC1_2_5	3325.6	3319.34	9	3600	3170.65	1	164	2262	0	3600
RC1_2_6	3300.7	3300.7	3	1028	3160.23	1	184	2637	0	3600
RC1_2_7	3177.8	3154.8	5	3600	3002.4	1	113	1915	0	3600
RC1_2_8	3060	3049.85	5	3600	2881.99	1	70	1395	0	3600
RC1_2_9	3073.3	3041.67	7	3600	2863.9	1	72	1443	0	3600
RC1_4_1	8522.9	8481.66	5	3600	8193.9	1	113	1646	0	3600
RC1_4_10	7581.2	7511.6	1	3600	6142.39	1	6	98	0	3600
RC1_4_2	7878.2	7843.85	1	3600	6800.16	1	14	389	0	3600
RC1_4_3	7516.9	7454.57	1	3600	-	-	-	-	-	3600
RC1_4_4	7292.9	7206.25	1	3600	-	-	-	-	-	3600
RC1_4_5	8152.3	8101.4	1	3600	7567.36	1	45	1575	0	3600
RC1_4_6	8148	8092.64	1	3600	7554.08	1	47	1644	0	3600
RC1_4_7	7932.5	7884.28	1	3600	7192.01	1	29	1036	0	3600
RC1_4_8	7757.2	7687.88	1	3600	6652.32	1	12	586	0	3600
RC1_4_9	7717.7	7641.23	5	3600	6587.73	1	12	574	0	3600
RC1_6_1	16960.1	16846.82	1	3600	15997.1	1	43	1283	0	3600
RC1_6_10	15651.3	15455.46	1	3600	-	-	-	-	-	3600
RC1_6_2	15890.6	15715.75	1	3600	-	-	-	-	-	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	14277.6	1	14	920	0	3600
RC1_6_6	16473.3	16315.89	1	3600	14316.6	1	16	938	0	3600
RC1_6_7	16055.3	15929.19	3	3600	13377.0	1	10	422	0	3600

Table K.1: Continued.

Instance		VRPSolver			Column Elimination					
Name	UB	LB	Nodes	Time (s)	LB	CEIt	CESIt	CR	Cuts	Time (s)
RC1_6_8	15891.8	15689.09	1	3600	-	-	-	-	-	3600
RC1_6_9	15803.5	15611.36	1	3600	-	-	-	-	-	3600
RC1_8_1	29978.9	29723.6	1	3600	27854.3	1	25	1162	0	3600
RC1_8_10	28168.5	27725.26	1	3600	-	-	-	-	-	3600
RC1_8_2	28290.1	27880.5	1	3600	-	-	-	-	-	3600
RC1_8_5	29219.9	28903.94	5	3600	25291.2	1	12	478	0	3600
RC1_8_6	29194.2	28795.58	5	3600	23585.7	1	4	109	0	3600
RC1_8_7	28788.6	28400.17	3	3600	-	-	-	-	-	3600
RC1_8_8	28418.1	28007.4	1	3600	-	-	-	-	-	3600
RC1_8_9	28347.1	27992.74	1	3600	-	-	-	-	-	3600
RC2_2_1	2797.4	2797.4	1	196	2069.59	1	320	1480	0	3600
RC2_2_10	1989.2	1954.78	1	3600	931.879	1	109	189	0	3600
RC2_2_2	2481.6	2481.6	3	1808	835.921	1	52	79	0	3600
RC2_2_4	1854.8	1839.25	1	3600	-	-	-	-	-	3600
RC2_2_5	2491.4	2491.4	1	684	1473.23	1	171	606	0	3600
RC2_2_6	2495.1	2495.1	1	713	1457.69	1	195	700	0	3600
RC2_2_7	2287.7	2284.42	5	3600	1218.47	1	143	421	0	3600
RC2_2_8	2151.2	2151.2	1	2476	1092.23	1	134	279	0	3600
RC2_2_9	2086.6	2059.93	1	3600	1121.09	1	149	239	0	3600
RC2_4_1	6147.3	6141.13	1	3600	3494.73	1	109	613	0	3600
RC2_4_2	5407.5	5328.56	1	3600	-	-	-	-	-	3600
RC2_4_5	5392.3	5369.86	1	3600	2234.53	1	51	178	0	3600
RC2_4_6	5324.6	5253.47	1	3600	2184.43	1	48	196	0	3600
RC2_4_7	4987.8	4848.73	1	3600	-	-	-	-	-	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	4444.29	1	31	218	0	3600

Table K.1: Continued.

Appendix L

Multicoloring Results

Table L.1: A comparison of the performance of the branch-and-price algorithm created by Gualandi and Malucelli (2012) and that of column elimination for solving the COG instances created in Gualandi and Malucelli (2012).

Instance				GM			Column Elimination				
Name	n	m	ω	LB	UB	Time (s)	LB	UB	CElt	CR	Time (s)
COG-10teams	3200	124480	73	-	-	3600	71	1600	25	3713	3373
COG-air04	17808	2121648	377	377	377	1.8	377	377	2	0	6
COG-air05	14390	2527253	413	-	-	3600	1	5295	0	0	2117
COG-atlanta-ip	8124	9250	15	15	15	1844	15	15	2	17	1
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**	141	581	139
COG-mkc	10394	154870	169	169	169	0.1	169	169	2	0	1
COG-mod011	192	336	12	12	13	3600	12	13	61	2395	3266
COG-mzzv11	19942	257012	101	101	101	0.1	101	101	2	0	5
COG-mzzv42z	18806	225687	91	91	91	0.1	91	91	2	0	4
COG-net12	3202	4835	17	17	17	1301	17	17	2	17	0
COG-nsrand-ipx	13240	69510	30	-	-	3600	30	**30**	2	0	5
COG-opt1217	1536	6528	26	-	-	3600	26	**26**	2	0	13
COG-rd-rplusc-21	904	11785	109	109	109	0	109	109	2	0	0
COG-rout	560	2940	30	30	32	3600	30	**30**	2	0	0
COG-swath	12480	958000	317	-	-	3600	-	-	-	-	3600

Appendix M

PDPTW Results

Table M.1: Comparing the performance of the dual ascent method from Baldacci et al. (2011a) and column elimination for solving some 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	10	1	7	3600
LC1_2_10	2741.6	2741.6	2741.6	137.1	-	3600	2389.82	1	324	6034	3600
LC1_2_2	2764.6	2764.6	2764.6	21.5	-	3600	2757.11	74	126	2608	3600
LC1_2_3	2772.2	2772.2	2772.2	114.9	-	3600	2499.26	1	124	2080	3600
LC1_2_4	2661.4	2395.8	2661.4	454.2	-	3600	-	-	-	-	3600
LC1_2_5	2702.0	2702.0	2702.0	4.8	-	3600	2702.05	10	13	176	130
LC1_2_6	2701.0	2701.0	2701.0	7.4	-	3600	2701.04	10	24	337	129
LC1_2_7	2701.0	2701.0	2701.0	7.7	-	3600	2701.04	9	23	311	130
LC1_2_8	2689.8	2689.8	2689.8	16.0	-	3600	2673.18	5	1053	6399	3600
LC1_2_9	2724.2	2724.2	2724.2	55.3	-	3600	2606.42	1	638	11183	3600
LR1_2_1	4819.1	4819.1	4819.1	1.6	-	3600	4819.12	18	1	416	75
LR1_2_10	3386.3	3386.3	3386.3	1376.7	-	3600	2614.42	1	342	3346	3600
LR1_2_2	4093.1	4093.1	4093.1	20.6	-	3600	3868.42	1	176	2011	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	4170.29	6	1309	8835	3600
LR1_2_6	3763.0	3763.0	3763.0	180.9	-	3600	3256.39	1	250	3008	3600
LR1_2_7	3112.9	2761.8	3112.9	1320.4	-	3600	-	-	-	-	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	3590.42	1	789	9524	3600
LRC1_2_1	3606.1	3606.1	3606.1	3.1	-	3600	3530.91	3	1711	12737	3600
LRC1_2_10	2837.5	2335.5	2837.5	217.4	-	3600	2146.89	1	379	4560	3600
LRC1_2_2	3292.4	3292.4	3292.4	322.3	-	3600	2778.25	1	226	3014	3600

Instance		BBM			VRPSolver		Column Elimination				
Name	UB	LB	UB	Time (s)	LB	Time (s)	LB	CElt	CESlt	CR	Time (s)
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	3021.66	1	994	13568	3600
LRC1_2_6	3360.9	3360.9	3360.9	7.0	-	3600	2750.99	1	183	2169	3600
LRC1_2_7	3317.7	3317.7	3317.7	408.2	-	3600	2658.79	1	861	10800	3600
LRC1_2_8	3086.5	3086.5	3086.5	1562.7	-	3600	2339.36	1	611	7051	3600
LRC1_2_9	3053.8	3053.8	3053.8	1757.2	-	3600	2340.17	1	556	6441	3600

Table M.1: Continued.

Table M.2: Comparing the performance of the dual ascent method from Baldacci et al. (2011a) and column elimination for solving some PDPTW instances by Li and Lim (2001) with 1000 locations.

Instance		BBM			VRPSolver		Column Elimination				
Name	UB	LB	UB	Time (s)	LB	Time (s)	LB	CElt	CESlt	CR	Time (s)
LC1_10_1	42488.66	42488.7	42488.7	79.5	-	3600	42432.6	2	257	2946	3600
LC1_10_5	42477.4	42477.4	42477.4	118.7	-	3600	39046.4	1	28	2561	3600
LR1_10_1	56744.91	56744.9	56744.9	233.1	-	3600	36732.4	1	9	613	3600
LR1_10_5	59053.68	52536.3	52901.3	4068.8	-	3600	41026.9	1	77	4069	3600
LRC1_10_1	49111.78	48398.8	48666.5	2533.3	-	3600	31414.6	1	37	2895	3600
LRC1_10_5	50323.04	38177.8	49287.1	1650.3	-	3600	-	-	-	-	3600

Table M.3: 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	CElt	CESlt	CR	Time (s)
LC1_4_1	7152.06	**7152.06**	8	15	217	50
LC1_4_2	8007.79	4980.79	1	4	433	3600
LC1_4_5	7150.0	**7150.0**	9	32	609	477
LC1_4_6	7154.02	**7154.02**	19	73	1867	3295
LC1_4_7	7149.43	7119.88	3	120	2573	3600
LC1_4_8	8305.42	6941.46	1	293	8755	3600
LC1_4_9	7451.2	5529.08	1	19	1149	3600
LC1_6_1	14095.64	14095.6	8	56	741	3600
LC1_6_5	14086.3	**14086.3**	8	91	2140	1620
LC1_6_6	14090.79	14002.8	1	168	4464	3600
LC1_6_7	14083.76	13443.7	1	44	2197	3600
LC2_2_1	1931.44	**1931.44**	37	54	494	300
LC2_2_10	1817.45	1697.61	1	496	3168	3600
LC2_2_2	1881.4	1839.51	1	231	1415	3600
LC2_2_3	1844.33	1605.18	1	90	458	3600
LC2_2_4	1767.12	1320.68	1	46	118	3600
LC2_2_5	1891.21	1852.14	5	389	2845	3600
LC2_2_6	1857.78	1794.7	2	854	6119	3600
LC2_2_7	1850.13	1804.02	1	711	4738	3600
LC2_2_8	1824.34	1740.64	1	551	3798	3600
LC2_2_9	1854.21	1744.24	1	527	3682	3600
LC2_4_1	4116.33	**4116.33**	12	113	1123	1555
LC2_4_10	3828.44	3302.42	1	83	809	3600
LC2_4_2	4144.29	3800.05	1	104	1109	3600

Instance		Column Elimination				
Name	UB	LB	CEIt	CESIt	CR	Time (s)
LC2_4_5	4030.63	3832.91	1	294	3675	3600
LC2_4_6	3900.29	3637.58	1	157	1858	3600
LC2_4_7	3962.51	3566.45	1	110	1189	3600
LC2_4_8	3844.45	3507.99	1	133	1571	3600
LC2_4_9	4188.93	3337.95	1	66	628	3600
LC2_6_1	7977.98	7741.69	1	189	2328	3600
LC2_6_10	7946.6	5481.81	1	28	278	3600
LC2_6_2	9900.48	6848.34	1	52	543	3600
LC2_6_5	9051.53	7226.97	1	183	3316	3600
LC2_6_6	8775.55	6832.82	1	81	1368	3600
LC2_6_7	9376.58	6266.68	1	37	391	3600
LC2_6_8	7579.63	6516.17	1	64	919	3600
LC2_6_9	8714.22	6173.42	1	41	488	3600
LR1_4_1	10639.75	10588.1	3	859	8357	3600
LR1_4_10	8192.65	4713.16	1	32	1336	3600
LR1_4_5	11374.06	8951.35	1	469	8829	3600
LR1_4_9	9859.47	7249.08	1	203	4032	3600
LR1_6_1	22821.65	21653.7	1	289	8766	3600
LR1_6_5	23623.52	17492.2	1	192	6161	3600
LR1_6_9	21835.87	13748.3	1	70	2195	3600
LR2_2_1	4073.1	3219.89	1	173	858	3600
LR2_2_10	3254.83	1602.35	1	62	55	3600
LR2_2_2	3796.0	1026.3	1	35	16	3600
LR2_2_5	3438.39	2179.08	1	93	209	3600
LR2_2_6	4457.95	1039.18	1	24	4	3600
LR2_2_9	3922.11	1912.56	1	84	110	3600
LR2_4_1	9726.88	5366.15	1	72	212	3600

Table M.3: Continued.

Instance		Column Elimination				
Name	UB	LB	CEIt	CESIt	CR	Time (s)
LR2_4_5	9894.46	2878.01	1	28	12	3600
LR2_4_9	7926.07	2536.86	1	26	30	3600
LR2_6_1	21759.33	7190.63	1	26	1	3600
LRC1_4_1	9124.52	7711.6	1	213	4997	3600
LRC1_4_10	7064.36	3579.29	1	8	1189	3600
LRC1_4_5	8847.4	6647.12	1	347	8422	3600
LRC1_4_6	8394.47	6391.08	1	351	8586	3600
LRC1_4_7	8037.87	5685.65	1	214	5303	3600
LRC1_4_8	7930.15	4882.51	1	111	3017	3600
LRC1_4_9	8004.24	4876.15	1	104	2846	3600
LRC1_6_1	18288.9	14261.5	1	149	5428	3600
LRC2_2_1	3595.18	2021.66	1	257	1219	3600
LRC2_2_2	3158.25	803.983	1	44	37	3600
LRC2_2_5	2776.93	1461.8	1	157	482	3600
LRC2_2_6	2707.96	1380.48	1	171	527	3600
LRC2_2_7	3010.68	1250.51	1	138	338	3600
LRC2_2_8	2399.89	1151.38	1	142	254	3600
LRC2_2_9	2208.49	1197.57	1	148	231	3600
LRC2_4_1	9738.95	3489.16	1	108	575	3600
LRC2_4_5	7309.54	2246.14	1	48	107	3600
LRC2_4_6	6337.08	1851.51	1	21	24	3600
LRC2_4_7	6292.23	1863.73	1	36	49	3600

Table M.3: Continued.

Bibliography

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows*. Prentice Hall, 1993.

Akers. Binary decision diagrams. *IEEE Transactions on computers*, 100(6):509–516, 1978.

H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 118–132. Springer, 2007.

K. M. Anstreicher and L. A. Wolsey. Two "well-known" properties of subgradient optimization. *Mathematical Programming*, 120(1):213–220, 2009.

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

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

P. Augerat, J.-M. Belenguer, E. Benavent, A. Corbéran, and D. Naddef. Separating capacity constraints in the cvrp using tabu search. *European Journal of Operational Research*, 106(2-3):546–557, 1998.

E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, pages 37–60. Springer, 1980.

E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19 (1):1–9, 1996.

R. Baldacci, N. Christofides, and A. Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115(2):351–385, 2008.

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

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

M. L. Balinski and R. E. Quandt. On an Integer Program for a Delivery Problem. *Operations Research*, 12 (2):300–304, 1964.

M. L. Balinski and A. W. Tucker. Duality theory of linear programs: A constructive approach with applications. *Siam Review*, 11(3):347–377, 1969.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46(3):316–329, 1998.

E. M. Beale. Branch and bound methods for mathematical programming systems. In *Annals of Discrete Mathematics*, volume 5, pages 201–219. Elsevier, 1979.

J. C. Bean, J. R. Birge, and R. L. Smith. Aggregation in dynamic programming. *Operations Research*, 35 (2):215–220, 1987.

J. C. Beck and E. C. Freuder. Simple rules for low-knowledge algorithm selection. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 50–64. Springer, 2004.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

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

D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Variable Ordering for the Application of BDDs to the Maximum Independent Set Problem. In *Proceedings of CPAIOR*, volume 7298 of *LNCS*, pages 34–49. Springer, 2012a.

D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. N. Hooker. Variable ordering for the application of bdds to the maximum independent set problem. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 34–49. Springer, 2012b.

D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Optimization Bounds from Binary Decision Diagrams. *INFORMS Journal on Computing*, 26(2):253–268, 2014a.

D. Bergman, A. A. Cire, W.-J. van Hoeve, and T. Yunes. Bdd-based heuristics for binary optimization. *Journal of Heuristics*, 20:211–234, 2014b.

D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Springer, 2016a.

D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. N. Hooker. *Decision diagrams for optimization*. Springer, 2016b.

D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016c.

T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

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

D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

N. Boland, J. Dethridge, and I. Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1):58–68, 2006.

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

R. R. Bouckaert, E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald, and D. Scuse. Weka manual for version 3-9-1. *University of Waikato, Hamilton, New Zealand*, 2016.

S. Boyd, L. Xiao, and A. Mutapcic. Subgradient methods. *Lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004:2004–2005, 2003.

D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293– 318, 1992.

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

Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau. Improving Optimization Bounds Using Machine Learning: Decision Diagrams Meet Deep Reinforcement Learning. In *Proceedings of AAAI*, pages 1443–1451. AAAI Press, 2019.

M. P. Castro, A. A. Ciré, and J. C. Beck. Decision Diagrams for Discrete Optimization: A Survey of Recent Advances. *INFORMS Journal on Computing*, 34(4):2271–2295, 2022.

Z.-L. Chen and W. B. Powell. Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing*, 11(1):78–94, 1999.

N. Christofides, A. Mingozzi, and P. Toth. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical programming*, 20(1):255–282, 1981a.

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

V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete mathematics*, 4(4): 305–337, 1973.

A. A. Ciré and J. N. Hooker. The Separation Problem for Binary Decision Diagrams. In *Proceedings of ISAIM*, 2014.

A. A. Ciré and W. J. van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61(6):1411–1428, 2013.

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

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

J. C. Culberson and F. Luo. Exploring the k-colorable landscape with iterated greedy. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge*, 26:245–284, 1996.

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

G. B. Dantzig and P. Wolfe. The decomposition algorithm for linear programs. *Econometrica: Journal of the Econometric Society*, pages 767–778, 1961.

G. B. Dantzig, A. Orden, P. Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

M. P. de Aragao and E. Uchoa. Integer program reformulation for robust branch-and-cut-and-price algorithms. In *Mathematical program in rio: a conference in honour of nelson maculan*, pages 56–61, 2003.

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

G. Desaulniers, F. Lessard, and A. Hadjar. Tabu search, partial elementarity, and generalized k-path inequalities for the vehicle routing problem with time windows. *Transportation Science*, 42(3):387–404, 2008.

J. Desrosiers, F. Soumis, and M. Desrochers. Routing with time windows by column generation. *Networks*, 14(4):545–565, 1984.

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

Ö. Ergun and J. B. Orlin. A dynamic programming methodology in very large scale neighborhood search applied to the traveling salesman problem. *Discrete Optimization*, 3(1):78–85, 2006.

L. F. Escudero, M. Guignard, and K. Malik. A lagrangian relax-and-cut approach for the sequential ordering problem with precedence relationships. *Annals of Operations Research*, 50:219–237, 1994.

M. Fischetti and P. Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37(2):319–328, 1989.

M. L. Fisher. Optimal solution of scheduling problems using lagrange multipliers: Part i. *Operations Research*, 21(5):1114–1127, 1973.

M. L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management science*, 27(1):1–18, 1981.

R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345–345, 1962.

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

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

E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 29(1):24–32, 1982.

S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 348–356, 1987.

M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, 1993.

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

M. Gagliolo and J. Schmidhuber. Algorithm portfolio selection as a bandit problem with unbounded losses. *Annals of Mathematics and Artificial Intelligence*, 61(2):49–86, 2011.

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

A. M. Geoffrion. Lagrangian Relaxation for Integer Programming. *Mathematical Programming Study 2*, pages 82–114, 1974.

X. Gillard and P. Schaus. Large Neighborhood Search with Decision Diagrams. In L. D. Raedt, editor, *Proceedings of IJCAI*, pages 4754–4760. ijcai.org, 2022.

P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research*, 9(6):849–859, 1961.

C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.

R. E. Gomory. Solving linear programming problems in integers. *Combinatorial analysis*, 10(211-215):25, 1960.

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

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

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

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations research*, 18(6):1138–1162, 1970.

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

J. Homberger and H. Gehring. Two evolutionary metaheuristics for the vehicle routing problem with time windows. *INFOR: Information Systems and Operational Research*, 37(3):297–318, 1999.

Y. N. Hoogendoorn and K. Dalmeijer. Resource-robust valid inequalities for vehicle routing and related problems. *arXiv preprint arXiv:2311.04825*, 2023.

J. N. Hooker. Decision diagrams and dynamic programming. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 94–110. Springer, 2013.

D.-I. M. Horn. *Advances in Search Techniques for Combinatorial Optimization: New Anytime A Search and Decision Diagram Based Approaches*. PhD thesis, Technische Universität Wien, 2021.

J. Huang and A. Darwiche. Dpll with a trace: From sat to knowledge compilation. In *IJCAI*, volume 5, pages 156–162, 2005.

T. Ibaraki and Y. Nakamura. A dynamic programming method for single machine scheduling. *European Journal of Operational Research*, 76(1):72–82, 1994.

S. Irnich and G. Desaulniers. Shortest Path Problems with Resource Constraints. In *Column Generation*, pages 33–65. Springer, 2005.

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

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

D. S. Johnson and M. A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey. *50 Years of integer programming 1958-2008: From the early years to the state-of-the-art*. Springer Science & Business Media, 2009.

A. Karahalios and W. van Hoeve. Column Elimination for Capacitated Vehicle Routing Problems. In A. A. Ciré, editor, *Proceedings of CPAIOR*, volume 13884 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2023a.

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

A. Karahalios and W.-J. van Hoeve. Column elimination for capacitated vehicle routing problems. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 35–51. Springer, 2023b.

A. Karahalios and W.-J. van Hoeve. Column elimination: An iterative approach to solving integer programs. *arxiv*, 2024.

N. Karmarkar, M. G. Resende, and K. Ramakrishnan. An interior point algorithm to solve computationally difficult set covering problems. *Mathematical Programming*, 52:597–618, 1991.

R. M. Karp and M. Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.

D. Kowalczyk, R. Leus, C. Hojny, and S. Røpke. A Flow-Based Formulation for Parallel Machine Scheduling Using Decision Diagrams A flow-based formulation for parallel machine scheduling using decision diagrams. *INFORMS Journal on Computing*, (Published Online), 2024.

R. Kuroiwa and J. C. Beck. Domain-Independent Dynamic Programming, 2024. arXiv preprint arXiv:2401.13883.

Y.-T. Lai, M. Pedram, and S. B. Vrudhula. Evbdd-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):959–975, 1994.

C.-Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

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

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

A. Lodi, M. Milano, and L.-M. Rousseau. Discrepancy-based additive bounding for the alldifferent constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 510–524. Springer, 2003.

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

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

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

A. Lucena. Lagrangian relax-and-cut algorithms. *Handbook of Optimization in Telecommunications*, pages 129–145, 2006.

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

J. Lysgaard, A. Letchford, and R. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming, Series A*, 100:423–445, 2004.

R. Macedo, C. Alves, J. V. de Carvalho, F. Clautiaux, and S. Hanafi. Solving the vehicle routing problem with time windows and multiple routes exactly using a pseudo-polynomial model. *European Journal of Operational Research*, 214(3):536–545, 2011.

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

C. U. Manual. Ibm ilog cplex optimization studio. *Version*, 12(1987-2018):1, 1987.

S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

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

J. E. Mitchell. Fixing variables and generating classical cutting planes when using an interior point branch and cut method to solve integer programming problems. *European Journal of Operational Research*, 97(1):139–148, 1997.

S. Mouthuy, P. V. Hentenryck, and Y. Deville. Constraint-based very large-scale neighborhood search. *Constraints*, 17:87–122, 2012.

N. Musliu and M. Schwengerer. Algorithm selection for the graph coloring problem. In *International Conference on Learning and Intelligent Optimization*, pages 389–403. Springer, 2013.

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

G. L. Nemhauser and S. Park. A polyhedral approach to edge coloring. *Operations Research Letters*, 10(6): 315–322, 1991.

P. R. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120 (1-3):197–207, 2002.

M. Padberg and G. Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations research letters*, 6(1):1–7, 1987.

D. Pecin, C. Contardo, G. Desaulniers, and E. Uchoa. New enhancements for the exact solution of the vehicle routing problem with time windows. *INFORMS Journal on Computing*, 29(3):489–502, 2017a.

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

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

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

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

B. T. Polyak. Minimization of unsmooth functionals. *USSR Computational Mathematics and Mathematical Physics*, 9(3):14–29, 1969.

B. T. Polyak. Subgradient methods: a survey of soviet research. *Nonsmooth optimization*, pages 5–29, 1978.

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

S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

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

R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.

I. Rudich, Q. Cappart, and L.-M. Rousseau. Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams. *Journal of Artificial Intelligence Research*, 77:1489–1538, 2023.

R. Sadykov, F. Vanderbeck, A. A. Pessoa, I. Tahiri, and E. Uchoa. Primal heuristics for branch and price: The assets of diving methods. *INFORMS J. Comput.*, 31(2):251–267, 2019.

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

P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In M. J. Maher and J. Puget, editors, *Proceedings of CP*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998.

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

A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *1990 IEEE international conference on computer-aided design*, pages 92–93. IEEE Computer Society, 1990.

Z. Tang. *Theoretical and Computational Methods for Network Design and Routing*. PhD thesis, Carnegie Mellon University, 2021.

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

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

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

J. Valério de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86(0):629–659, 1999.

J. M. van Den Akker, J. A. Hoogeveen, and S. L. van de Velde. Parallel machine scheduling by column generation. *Operations research*, 47(6):862–872, 1999.

W. van Hoeve. Graph coloring lower bounds from decision diagrams. In D. Bienstock and G. Zambelli, editors, *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*, pages 405–418. Springer, 2020a.

W.-J. van Hoeve. Graph Coloring Lower Bounds from Decision Diagrams. In *Proceedings of IPCO*, volume 12125 of *LNCS*, pages 405–419. Springer, 2020b.

W.-J. van Hoeve. Graph Coloring Lower Bounds from Decision Diagrams. In *Proceedings of IPCO*, volume 12125 of *LNCS*, pages 405–419. Springer, 2020c.

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

W.-J. van Hoeve. An Introduction to Decision Diagrams for Optimization. In *INFORMS TutORials in Operations Research*. INFORMS, 2024.

W.-J. van Hoeve and Z. Tang. Column "Elimination": Dual Bounds From Decision Diagram-based Route Relaxations. In *INFORMS Computing Society Conference*, 2022.

F. Vanderbeck. Implementing Mixed Integer Column Generation. In G. Desaulniers, J. Desrosiers, and M. Solomon, editors, *Column Generation*, pages 331–358. Springer, 2005.

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

D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM monographs on discrete mathematics and applications. Society for Industrial and Applied Mathematics, 2000.

H. Weyl. The elementary theory of convex polyhedra. *Contributions to the Theory of Games*, 1(24):3–18, 1950.

L. A. Wolsey. *Integer programming*. John Wiley & Sons, 2020.

N. A. Wouda, L. Lan, and W. Kool. Pyvrp: A high-performance vrp solver package. *INFORMS Journal on Computing*, 36(4):943–955, 2024.

L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.