Interval Scheduling with Economies of Scale

Christopher Muir Alejandro Toriello
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA 30332
chris dot muir at gatech dot edu, atoriello at isye dot gatech dot edu

July 27, 2022

Abstract

Motivated by applications in cloud computing, we study interval scheduling problems exhibiting economies of scale. An instance is given by a set of jobs, each with start time, end time, and a function representing the cost of scheduling a subset of jobs on the same machine. Specifically, we focus on the max-weight function and non-negative, non-decreasing concave functions of total schedule weight. The goal is a partition of the jobs that minimizes the total schedule cost, where overlapping jobs cannot be processed on the same machine. We propose a set cover formulation and a column generation algorithm to solve its linear relaxation. For the max-weight function, which is already NP-hard, we give a polynomial-time pricing algorithm; for the more general case of a function of the total weight, we have a pseudo-polynomial algorithm. To obtain integer solutions, we extend the column generation approach using branch-and-price. We computationally evaluate our methods on two different functions, using both random instances and instances derived from cloud computing data; our algorithm significantly outperforms known integer programming formulations (when these are available) and is able to provably optimize instances with hundreds of jobs.

1 Introduction

Interval scheduling, sometimes also called fixed interval scheduling, is a broad class of problems arising in operations research, computer science, production, scheduling, and logistics. Generally, interval scheduling problems task the decision maker with scheduling a set of jobs by assigning them to machines. Each job is defined by a fixed start and end time and each machine can only process one job at a time. A feasible solution is a partition of jobs into schedules such that jobs within the same schedule do not overlap in time. Various objectives are used in interval scheduling problems and are typically specific to the application.

Interval scheduling and its related problems have numerous applications. Some of these include channel assignment [23], fleet planning [28], bus scheduling [29], satellite scheduling [20, 33], and circuit design [34]. We are primarily interested in interval scheduling for its applications in the area of cloud scheduling. An important problem in cloud services is the allocation of virtual machine (VM) requests to machines. Each VM request has a start time, end time and a set of resource requirements, and the goal is to assign the VMs to machines such that each machine's

capacity is never exceeded. Such allocation problems lead to a temporal bin packing problem, with typical objectives including either to minimize the total number of machines required to process all VMs [12, 19], or to minimize some operating cost, e.g. power consumption [3]. Interval scheduling is a special case in which each machine can only process one VM at a time, as when managing requests for very large services, such as certain machine learning systems. When VM's cannot run in parallel on the same machine, the assignment component of the problem is somewhat simplified, allowing for the consideration of more complex and/or realistic objectives; this is one of our main motivations.

In this work, we study the cost minimization form of interval scheduling. Given some function that computes a cost for each schedule, the decision maker seeks a partition of the jobs that minimizes the sum of schedule costs. We study cost functions exhibiting economies of scale; specifically, we focus on the max-weight function and concave functions of total schedule weight. The max-weight function models a scenario in which different sizes of servers are available, e.g, servers with different CPU capacities. The cost of a server is proportional to its size; therefore, the cost of running a group of VMs together is proportional to the most resource-intensive VM in the group. Functions of total weight are flexible enough to model many important applications, e.g., rent-or-buy scheduling [15]. Specifically, concave functions capture the increased efficiency of scheduling jobs on the same machine. In the context of VM scheduling, one example of this efficiency is the reduction in start-up and shut-down costs.

Interval scheduling with the max-weight function and many concave functions is known to be NP-Hard, and despite various theoretical results, work on exact solution approaches is limited. With this motivation, we propose an approach based on column generation for a set covering formulation. We summarize our primary contributions as follows:

1. To the best of our knowledge, we are the first to address the exact optimization of certain NP-Hard variants of interval scheduling.

2. We propose efficient pricing algorithms for the cases of a max-weight function and more general functions based on a schedule's cumulative weight.

3. For the specific case of the max-weight function on path instances, which is known to be polynomially solvable, existing formulations and our proposed set cover formulation are fractional; we provide a tight linear programming formulation. (In a path instance, each job overlaps only with its immediate predecessor and successor.)

4. We present a detailed computational study demonstrating the effectiveness of our approach; in particular, we are able to exactly optimize instances with up to several hundred jobs.

The outline of the paper is as follows. In the next section we provide an overview of the relevant literature. In Section 3, we give a formal description of our model, provide the relevant preliminaries, and introduce our set covering formulation. In Section 4, we describe our column generation algorithm, present details of our pricing algorithms, and extend our approach to

branch-and-price. In Section 5, we study the special case of the max-weight schedule cost on a path. Section 6 details the results of our computational study, and Section 7 concludes.

2 Literature Review

The literature on interval scheduling is extensive. In the most basic variant, the goal is to minimize the number of machines required to process all jobs. This can be solved in linear time using a first-fit rule [32]. Much of the interval scheduling literature focuses on a related problem that asks, for some integer k, what is the maximum cardinality or maximum-weight set of jobs that can be processed on k machines. [9] provide a $O(n+k)$ algorithm for the cardinality case and a $O(nS(n))$ algorithm for the weighted case, where $S(n)$ is the running time of a shortest-path algorithm on a graph with $O(n)$ edges. [2] propose a linear programming (LP) approach, making use of a graph-theoretic interpretation of interval scheduling: For a given instance, its *conflict graph* is an interval graph constructed by creating a node for each job and adding an edge between two nodes when the corresponding jobs overlap. The incidence matrix formed using the maximal cliques in the interval graph has the consecutive ones property and is therefore totally unimodular; this leads to an integral linear program capable of solving the weighted k-machine problem. The survey [27] provides a detailed overview of common variants of k-machine interval scheduling. Example variants include allowing for machine downtime [7], non-identical machines [2, 31], and jobs requiring multiple processing intervals [4].

Closely related to our work is the literature on submodular coloring, its complement, submodular clique partitioning, and submodular interval scheduling, which is submodular coloring restricted to interval graphs. In [10], the authors study value-monotone submodular functions and provide a 5-approximation algorithm for interval graphs and a 7-approximation for perfect graphs when the cost function is a non-decreasing weight-defined concave function. [21] explore a broader class of submodular functions, referred to as value-polymatroidal; the authors provide a dynamic programming algorithm for co-interval graphs. [16] present a robust coloring algorithm and provide a 4-approximation algorithm for interval graphs for any concave function. Additionally, [16] prove NP-Hardness for coloring interval graphs with strictly concave functions. This result is based on a proof that minimum-entropy coloring is NP-Hard for interval graphs [8]. [14] study coloring in general graphs with the max-weight function, in which the cost of a color class is equal to weight of the highest-weight node contained in the class, and prove that it is NP-Hard for interval graphs.

Methodologically, our work leverages column generation, which is used to solve linear programs with exponentially many variables. Such problems often arise when solving the linear relaxation of exponentially-sized reformulations of integer programs (IP). These reformulations are desirable because they typically provide stronger dual bounds compared to compact formulations, and solutions to the relaxation can be used to generate heuristic primal solutions. For a general reference on column generation, we refer the reader to [13]. Finding integer solutions

to column generation models is challenging, as standard IP techniques like branch-and-bound are ill-suited. This motivates the development of the branch-and-price algorithm [5], which combines column generation with branch-and-bound. Branch-and-price algorithms have been used successfully in many applications; some examples related to our problem include graph coloring [30], max-weight coloring [17], sum coloring [11], robust coloring [22], partition coloring [18], clique partitioning with minimum size [26], and interval scheduling with a resource constraint [1].

3 Problem Statement and Preliminaries

We consider interval scheduling problems taking as input a set of n jobs $J = \{1, 2, 3, ..., n\}$, a vector of job weights $w \in \mathbb{Z}^n_+$, and a cost function $f : 2^J \to \mathbb{R}_+$, where 2^J denotes the power set of J. Each job $i \in J$ has a start-end time pair (s_i, e_i), $s_i, e_i \in \mathbb{Z}_+$, $s_i < e_i$. We assume without loss of generality that jobs are ordered by non-decreasing start times, then by end times if necessary. We refer to subsets $S \subseteq J$ as *schedules*, and S is a *feasible schedule* if jobs in S do not overlap in time, $[s_i, e_i] \cap [s_j, e_j] = \emptyset$ for $i, j \in S$. Each interval scheduling instance has a corresponding graphical representation $G = (N, E)$. This *conflict* graph is an interval graph constructed by creating a node for each $i \in J$ and adding an edge between nodes when their corresponding jobs overlap in time.

The decision maker is interested in partitioning J into disjoint feasible schedules $S_1, S_2, ..., S_k$ such that the combined cost of the schedules is minimized. Letting $\mathbb{S}$ be the set of all feasible schedules, the problem is

$$\min_{S_1,\ldots,S_k \in \mathbb{S}} \left\{ \sum_{\ell=1}^{k} f(S_\ell) : \quad \bigcup_{\ell=1}^{k} S_\ell = J; \quad S_\ell \cap S_m = \emptyset,\ \ell \neq m \right\}. \tag{1}$$

Problem (1) is equivalent to a minimum-cost coloring problem on the instance's conflict graph. We consider the following cost functions f:

1. Max-Weight: $f(S) = \max_{i \in S} w_i$
2. Non-Negative, Non-Decreasing, Concave: $f(S) = h\left(\sum_{i \in S} w_i\right)$ for some non-negative, non-decreasing concave function h

Both of these functions belong to the class of *value-polymatroidal* functions [21] and *value-monotone submodular* functions [10]. Additionally, both are known to be theoretically difficult; the max-weight function and many weight-defined concave functions make (1) NP-Hard [14, 16].

Problem (1) with the max-weight function is a special case of the general *max-weight coloring* problem restricted to interval graphs. The max-weight coloring problem has a natural mixed-integer linear programming (MILP) formulation: Given a graph $G = (N, E)$, let Δ be the maximum degree in G, and $\mathbb{C}$ its set of maximal cliques. If G is an interval graph, we additionally use $\mathbb{C}$ to represent the corresponding sets of jobs in the underlying scheduling instance. The max-weight

coloring problem is then formulated as

$$
\begin{aligned}
\min_{x,y} \ & \sum_{k=1}^{\Delta+1} y_k && && \text{(2a)}\\
\text{s.t.} \ & \sum_{i \in C} x_{i,k} \leq 1 && \forall C \in \mathbb{C}, \forall k \in \{1,2,..,\Delta+1\} && \text{(2b)}\\
& \sum_{k=1}^{\Delta+1} x_{i,k} = 1 && \forall i \in N && \text{(2c)}\\
& w_i x_{i,k} \leq y_k && \forall i \in N, \forall k \in \{1,2,..,\Delta+1\} && \text{(2d)}\\
& x_{i,k} \in \{0,1\} && \forall i \in N, \forall k \in \{1,2,..,\Delta+1\} && \text{(2e)}\\
& y_k \geq 0 && \forall k \in \{1,2,...,\Delta+1\}. && \text{(2f)}
\end{aligned}
$$

The variables $x_{i,k}$ denote the binary decision to place job i in the k-th schedule, and the variables y_k keep track of the maximum-weight job in the k-th schedule. This formulation makes use of the observation that an optimal solution would never require more than $\Delta + 1$ colors.

For general graphs, (2) may be exponentially large depending on the graph's number of maximal cliques. When restricted to interval graphs, the model is polynomial as the number of maximal cliques is $O(n)$. Despite this advantage, in practice modern MILP solvers struggle to solve (2) for interval scheduling instances with even a moderate number of jobs, in part because of its symmetry.

3.1 Set Covering Formulation

We propose a formulation for (1) with exponentially many variables. Our model operates in the space of schedules, with each binary variable indicating whether to use a particular schedule. This yields

$$
\begin{aligned}
\min_{z} \ & \sum_{S \in \mathbb{S}} f(S) z_S && && \text{(3a)}\\
\text{s.t.} \ & \sum_{S \in \mathbb{S}, S \ni i} z_S \geq 1 && \forall i \in J && \text{(3b)}\\
& z_S \in \{0,1\} && \forall S \in \mathbb{S}. && \text{(3c)}
\end{aligned}
$$

We use $S \ni i$ to denote the requirement that schedule S contains i; the variable z_S represents the binary decision to use schedule S in the solution. Although feasible solutions should technically be partitions of the job set J, we relax the equality constraint to greater-than-or-equal in (3b), transitioning from a partition model into an equivalent covering model; the equivalence follows from the monotonicity of the function f. This helps improve the convergence of the column generation algorithm.

4 Solution Methodology

We solve the linear relaxation of (3) with column generation. The algorithm maintains a *restricted master problem* (RMP) that only contains a subset of the variables. Letting $\mathbb{S}_{\text{RMP}} \subseteq \mathbb{S}$ denote the set of schedules currently in the RMP, we obtain

$$\min_{z} \left\{ \sum_{S \in \mathbb{S}_{\text{RMP}}} f(S) z_S : \quad \sum_{S \in \mathbb{S}_{\text{RMP}}, S \ni i} z_S \geq 1,\ i \in J; \quad z_S \geq 0,\ S \in \mathbb{S}_{\text{RMP}} \right\}. \tag{4}$$

Given a particular set of schedules $\mathbb{S}_{\text{RMP}}$, we solve (4) and use the optimal dual multipliers to check if any schedule $S \notin \mathbb{S}_{\text{RMP}}$ has negative reduced cost; this is known as the *pricing problem*. If we find any such schedules, we add them to $\mathbb{S}_{\text{RMP}}$ and re-solve. Otherwise, the current solution is optimal for the linear relaxation of (3).

After solving (4), the pricing problem is

$$\max_{S \in \mathbb{S}} \left\{ \sum_{i \in S} \pi_i - f(S) \right\}, \tag{5}$$

where π_i is the dual multiplier of the covering constraint in (4) for job i. If the optimal value of (5) is positive, this corresponds to a schedule with negative reduced cost that can be added to $\mathbb{S}_{\text{RMP}}$. In the next two sub-sections, we respectively present methods for solving (5) when f is the max-weight function or any function of the total schedule weight.

4.1 Pricing: Max-Weight Function

Substituting $f(S) = \max_{i \in S} w_i$ in (5) yields the pricing problem

$$\max_{S \in \mathbb{S}} \left\{ \sum_{i \in S} \pi_i - \max_{i \in S} w_i \right\}. \tag{6}$$

Proposition 4.1. *Problem (6) can be solved in $O(nP(n))$ time, where $P(n)$ is the complexity of solving an interval packing problem with n jobs.*

Proof. Assume we fix an upper bound $\bar{w} \in \{w_1, w_2, ..., w_n\}$ on the weight of jobs we can put in a schedule; then (6) reduces to

$$\max_{S \in \mathbb{S}} \left\{ \sum_{i \in S} \pi_i : \quad w_i \leq \bar{w}, i \in S \right\}.$$

That is, the problem reduces to finding a schedule maximizing the sum of the π_i values, using only jobs with $w_i \leq \bar{w}$. If a solution S has $\max_{i \in S} w_i < \bar{w}$, the same schedule S is optimal for a smaller value of $\bar{w}$.

This is an *interval packing* problem, a special case of the *node packing* or *independent set* problem restricted to interval graphs [24]. As there are at most n distinct values for $\bar{w}$, (6) can be solved by

solving at most n interval packing problems. □

Proposition 4.1 implies that the complexity of (6) is determined by the complexity of the resulting interval packing problem. Interval packing is well known to be polynomially solvable. We present two algorithms for its solution, both of which we implement as part of our branch-and-price methodology. The first of these methods is based on linear programming; given dual prices π, we formulate the interval packing problem as

$$\max_{x}\left\{\sum_{i\in J}\pi_i x_i: \quad \sum_{i\in C} x_i \leq 1,\ \forall C\in\mathbb{C}; \quad x_i\geq 0,\ \forall i\in J\right\}.$$

For general graphs, this LP is not necessarily integral; however, for interval graphs the constraint matrix of the formulation satisfies the consecutive-ones property, and is therefore totally unimodular. This implies that an optimal extreme point is integer and yields the desired packing. Additionally, as the number of maximal cliques in an interval graph is $O(n)$, the LP can be solved efficiently using any standard linear programming algorithm.

A full round of pricing in our scheme requires solving multiple interval packing instances; each instance is specified by an upper bound $\bar{w}$ on the weight of jobs admissible to the packing. This upper bound can be incorporated into the LP by fixing $x_i = 0$ when w_i is greater than $\bar{w}$. A benefit of the linear programming approach is the ability to warm-start; we solve the sub-problems in increasing order with respect to the weight upper bound, and each LP can be warm-started from the previous optimal solution, as primal feasibility is maintained.

The second method to solve (6) is based on dynamic programming (DP). We define states $i\in\{0,1,2,...,n,n+1\}$, where state $i=1,2,..,n$ denotes that job $i\in J$ was the last job added to the schedule, and states $0, n+1$ serve as artificial start and stop points. At each state, the decision maker determines which job is next in the schedule, or ends the schedule. The action set at i is $A_i=\{j\in J: s_j\geq e_i\}\cup\{n+1\}$, with rewards $r_j=\pi_j$ for $j\in J$ and $r_{n+1}=0$. The initial state is 0 and its action set is $A_0 = J\cup\{n+1\}$. We use v_i to denote the optimal value at state i, that is, the maximum weight of an interval packing containing i and only including jobs $j\leq i$. We obtain the recursion

$$v_i=\max_{j\in A_i}\{r_j+v_j\}, \quad \forall i\in J\cup\{0\}, \qquad\qquad v_{n+1}=0. \tag{7}$$

The above DP has $\Theta(n)$ states and $O(n)$ actions per state, making the total complexity $O(n^2)$. While generally leading to a slower empirical running time than the LP method, the DP model more easily incorporates side constraints on the structure of feasible schedules; specifically, it allows merging and separation constraints that require or forbid certain jobs from running consecutively in a schedule. These constraints are important as they correspond to branching disjunctions in our branch-and-price algorithm described in Section 4.3.

It is possible to obtain a DP with linear complexity by instead making admit/reject decisions at each job state. The jobs are considered in arrival order and if a job is admitted the decision

maker receives the reward π_i and the state transitions to the next job that could be feasibly added to the schedule. It is easy to verify that an algorithm of this type has $O(n)$ running time, but it loses the ability to incorporate separation and merge constraints, the primary advantage of the DP approach over LP. We discuss further details on this type of DP below in Section 4.2.

Identifying the most improving column requires solving $O(n)$ interval packing problems; however, any sub-problem may produce an improving column. An option is to conduct *partial pricing* and terminate after finding any improving column. This typically trades a higher total iteration count for a lower time per iteration. Likewise, it is also possible to conduct a full round of pricing and return all improving columns instead of just the most improving. These implementation details can have a major impact on the running time and convergence of the column generation algorithm. See Section 6.1 for details on this design decision.

4.2 Pricing: Function of Total Weight

Next we consider the pricing problem (5) when f is a function of the total schedule weight. That is, for some function h, the cost is $f(S) = h(\sum_{i\in S} w_i)$. This includes the special case in which h is non-negative, non-decreasing and concave. This leads to the pricing problem

$$\max_{S\in\mathcal{S}} \left\{ \sum_{i\in S} \pi_i - h\left(\sum_{i\in S} w_i \right) \right\}.$$

We use DP to solve this problem: Define states (i, W) for $i \in \{0, 1, 2, ..., n+1\}$ and $W \in \{0, 1, 2, ..., \sum_{i\in J} w_i\}$, where state (i, W) indicates that i was just added to the schedule and the sum of the scheduled jobs' weights is W. As with (7), we define the action set at state (i, W) as $A_i = \{j \in J : s_j \geq e_i\} \cup \{n+1\}$ and $A_0 = J \cup \{n+1\}$. The reward for taking action $j \in J$ is $r_j = \pi_j$ and $r_{n+1} = 0$. The initial state is $(0, 0)$. Using $u_{i,W}$ to denote the optimal value at state (i, W), we get the recursion

$$u_{i,W} = \max_{j\in A_i} \{r_j + u_{j,W+w_i}\} \qquad \forall i \in J \cup \{0\}, \forall W = 0, 1, ..., \sum_{i\in J} w_i \tag{8a}$$

$$u_{n+1,W} = -h(W) \qquad \forall W = 0, 1, ..., \sum_{i\in J} w_i. \tag{8b}$$

The DP has $\Theta(n \sum_{i\in J} w_i)$ states and each state has $O(n)$ actions; therefore, it is pseudo-polynomial with complexity $O(n^2 \sum_{i\in J} w_i)$. If $\sum_{i\in J} w_i$ is a polynomial function of n, the complexity is polynomial, e.g. when h is a function of the schedule's cardinality. As before, an alternate DP formulation with lower complexity is possible, at the cost of less modeling power; we discuss this approach next.

Again, we define states (i, W) for $i \in J$ and $W \in \{0, 1, ..., \sum_{i\in J} w_i\}$, where i now indicates deciding whether to add i to the schedule or not. Specifically, the actions at (i, W) are to add i, receiving reward π_i and transitioning to state $(\eta_i, W + w_i)$, where $\eta_i = \min\{j \in J : s_j \geq e_i\}$, or to reject i, receiving no reward and transitioning to $(i+1, W)$. The initial state is $(0, 0)$. Letting $\sigma_{i,W}$

denote the optimal value at state (i, W), we have

$$\sigma_{i,W} = \max\{\pi_i + \sigma_{\eta_i, W+w_i}, \sigma_{i+1,W}\} \qquad \forall i \in \{1,2,3,...,n-1\}, \forall W = 0,1,..., \sum_{i \in J/\{n\}} w_i \tag{9a}$$

$$\sigma_{n,W} = \max(\pi_n - h(W + w_n), -h(W)) \qquad \forall W = 0,1,..., \sum_{i \in J/\{n\}} w_i. \tag{9b}$$

This DP has $\Theta(n \sum_{i \in J} w_i)$ states and two actions per state, so the total complexity is $\Theta(n \sum_{i \in J} w_i)$. This complexity is better than (8) by a factor of n, and is likely the faster pricing method when there are no separation or merge constraints, i.e. at the root node of a branch-and-price tree. Lastly, as with the max-weight function, it is possible to identify multiple improving columns. Both DP recursions include information on the best schedule starting from job j for all jobs $j \in J$; any such schedule may also provide an improving column.

4.3 Branch-and-Price Algorithm

We now extend the column generation approach to an exact algorithm for (3) by combining column generation and branch-and-bound. The LP at each node in the search tree is solved via column generation. Instead of branching directly on the variables in (3), we adopt a branching scheme common in the vehicle routing literature: We define arc variables $\theta_{i,j} \in [0, 1]$ for each pair $i, j \in J$, $i < j$, of non-overlapping jobs, representing the decision that j immediately follows i in some schedule. Given a fractional solution z^*, we define $\theta_{i,j}$ to equal to the sum of all z^*_S variables for S in which jobs i, j appear consecutively. Formally, $\theta_{i,j} = \sum_{S \in \mathbb{S}} \mathbb{1}_{\{(i,j) \in S\}} z^*_S$, where $\mathbb{1}_{\{(i,j) \in S\}}$ indicates that j immediately follows i in schedule S. For a solution with a fractional $\theta_{i,j}$, the branching decision partitions the solution space using the constraints $\theta_{i,j} = 1$ and $\theta_{i,j} = 0$. This is equivalent to partitioning the space based on merge and separation constraints: j must immediately follow i, or j cannot immediately follow i. Both recursions (7) and (8) can incorporate these branching constraints by updating their respective action sets. In the $\theta_{i,j} = 0$ branch, we eliminate the action j from states at i. For the $\theta_{i,j} = 1$ branch, j is the only action at i, and j is removed from the action set of all other states. We outline additional details of our branch-and-price implementation in Section 6.1.

4.4 Primal Heuristics

To complement the lower-bounds obtained from solving the linear relaxation of (3), we propose a constructive heuristic and local-search algorithm for both the max-weight and concave functions. We use a greedy packing heuristic to obtain an initial feasible solution. Starting from the initial set of jobs $U = J$ and an empty set of schedules $V = \emptyset$, we solve an interval packing problem to obtain a feasible schedule S, update the sets to $U \leftarrow U \setminus S$ and $V \leftarrow V \cup \{S\}$, and repeat until $U = \emptyset$, which is achieved when V corresponds to a feasible solution to (1). While not explicitly required, we find that the heuristic's performance improves using modified weights $\hat{w}_i = w_i^2$; this biases the

heuristic to prefer scheduling high-weight jobs together over scheduling many low-weight jobs.

We use a local search heuristic to improve the initial solution found by the greedy heuristic. Given a set of schedules V, the heuristic considers pairs of schedules $S_\ell, S_m \in V$ and searches for a pair of schedules S'_ℓ, S'_m such that $f(S'_\ell) + f(S'_m) < f(S_\ell) + f(S_m)$ and $S'_\ell \cup S'_m = S_\ell \cup S_m$. The local search repeatedly checks all pairs $S_\ell, S_m \in V$, updating as better schedules are found. The search continues until no pair results in an improved solution.

The local search algorithm requires solving an optimization problem for the pairwise schedule improvement. This is equivalent to the interval scheduling problem (1) restricted to instances with conflict graphs having cliques of size at most two, i.e., the conflict graphs are bipartite. For the max-weight function, this can be done directly using the formulation (2), limiting the number of schedules to two. This problem can be solved in polynomial time, as we establish with the following proposition.

Proposition 4.2. *Given schedules $S_\ell, S_m \subset J$, the problem*

$$\min\{\max_{i \in S'_\ell} w_i + \max_{i \in S'_m} w_i : S'_\ell \cup S'_m = S_\ell \cup S_m\}$$

can be solved in polynomial time.

Proof. First assume that the instance's conflict graph is connected. Ignoring symmetry, there is a single solution to this problem using two schedules. Start at the earliest arriving job, then begin assigning jobs in arrival order. By assumption the conflict graph's coloring number is two and this placement will produce a solution using two schedules, see [32]. To see this is the only solution, note that each job i's color is determined entirely by the color of earliest arriving job j that i overlaps. As the graph is connected, after choosing the color of the first arriving job the remaining decisions are fixed.

If the conflict graph has more than one connected component, an optimal solution is obtained by constructing one schedule S'_ℓ containing the schedule with minimum max-weight in each component and placing the remainder in a second schedule S'_m. By construction, this will create a solution in which $\max_{i \in S'_\ell} w_i$ is minimized and $\max_{i \in S'_m} w_i = \max_{i \in S_\ell \cup S_m} w_i$; there will always be at least one schedule with cost $\max_{i \in S_\ell \cup S_m} w_i$. Finally, the construction of the schedules can be done in $O(n)$ time and the result follows. □

For non-negative, non-decreasing concave functions, we use a similar argument as in Proposition 4.2.

Proposition 4.3. *Given schedules $S_\ell, S_m \subset J$, the problem*

$$\min\left\{h\left(\sum_{i \in S'_\ell} w_i\right) + h\left(\sum_{i \in S'_m} w_i\right) : S'_\ell \cup S'_m = S_\ell \cup S_m\right\},$$

where h is a non-negative, non-decreasing concave function, can be solved in polynomial time.

Proof. Consider the same algorithm as described in the proof of Proposition 4.2; instead of assigning to S'_ℓ the sub-schedules with minimum max-weight, assign the sub-schedule in each component with the larger total weight. By the same argument as before, this will create the schedule S'_ℓ with the maximum total weight. As each job must be scheduled, after placing the remaining jobs in S'_m, it will be the schedule of minimum total weight. Finally, as this solution maximizes the difference $h(\sum_{i \in S'_\ell} w_i) - h(\sum_{i \in S'_m} w_i)$, the concavity of h implies this solution minimizes $h(\sum_{i \in S'_\ell} w_i) + h(\sum_{i \in S'_m} w_i)$ and the result follows. □

5 Max-Weight Function on Paths

In this section, we focus on the max-weight function restricted to instances where the conflict graph is a path. Equivalently, each job $1 < i < n$ overlaps with its predecessor and successor, $i-1$ and $i+1$, and with no other jobs. Unlike the more general problem setting we consider, this special case is polynomially solvable [14]; however, there are instances in which the linear relaxations of both (2) and (3) are fractional. Motivated by this discrepancy, we present a tight LP formulation for this special case that leverages the algorithm in [14] and the fact that an optimal solution never uses more than three schedules.

The intuition behind the algorithm is the following. Each schedule is defined by its highest-weight job, and one of these is always the maximum-weight job, $f(S_1) = \max_{k \in J} w_k$. Some job $i \in J$ defines the second schedule by having its highest-weight job, $f(S_2) = w_i$, and the third schedule then satisfies $f(S_3) \leq f(S_2)$. This implies that $j \in S_1$ if $w_j > w_i$; furthermore, for any two jobs $j,k \in S_1$, $j < k$ with an even number of jobs in between, a feasible solution must have $\ell \in S_3$ for some job $\ell \in \{j+1, j+2, ..., k-2, k-1\}$; we use $\ell \in (j,k)$ to denote the latter condition. (Conversely, if there is an odd number of jobs between j and k, they can be assigned in alternating fashion to S_1 and S_2.) Denote job i's minimal even pairs as

$$E(i) = \{(j,k) : \quad j,k \in J; \quad w_j, w_k > w_i; \quad j < k; \quad k - j = 1 \mod 2; \quad w_\ell \leq w_i,\ \ell \in (j,k)\}.$$

We illustrate idea with the six-job example in Figure 1(a), with weight vector $w = (6,1,2,5,3,4)$.

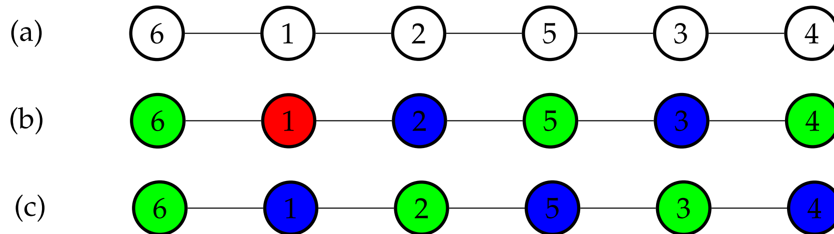

Figure 1: Example path instance with six jobs.

Suppose $f(S_2) = w_5 = 3$; then S_1 must contain the first, fourth and sixth jobs, as $w_1, w_4, w_6 > 3$.

Clearly, we also assign the fifth job to S_2. However, we cannot assign both the second and third jobs to S_2, because they are adjacent; one of these must go in S_3, so we place the second job there, since it has the lower weight of the two. The resulting schedule is illustrated in Figure 1(b).

Suppose instead that $f(S_2) = w_4 = 5$. In this case, S_1 must contain the first job. However, this also means there are no even pairs, and we can simply assign jobs to S_1 and S_2 in alternating fashion. This schedule is illustrated in Figure 1(c). For this instance, these are essentially the only options: If we choose $f(S_2) \leq 2$, we force S_1 to contain adjacent jobs, which is infeasible. This can also be interpreted as having even pairs of the form $(j, j+1)$, which have no interior element and thus cannot include a job in S_3. Finally, if we choose $f(S_2) \in \{4, 6\}$, we are forced into one of the previously illustrated schedules.

Formally, there is an optimal solution satisfying the following conditions [14, Theorem 7]:

C.1 The solution uses at most three schedules, S_1, S_2, S_3, where possibly $S_3 = \emptyset$.

C.2 $\max_{k \in J} w_k = f(S_1) \geq f(S_2) \geq f(S_3)$.

C.3 Given $f(S_2) = w_i$, S_3 contains exactly one job $\ell \in (j, k)$ from each $(j, k) \in E(i)$.

Based on this, we formulate

$$\min \sum_{i \in J} (w_i x_i + \lambda_i) \tag{10a}$$

$$\text{s.t.} \sum_{i \in J} x_i = 1 \tag{10b}$$

$$x_i = \sum_{j < \ell < k} y_{i,\ell} \qquad \forall i \in J, \forall (j,k) \in E(i) \tag{10c}$$

$$\lambda_i \geq \sum_{j < \ell < k} w_\ell y_{i,\ell} \qquad \forall i \in J, \forall (j,k) \in E(i) \tag{10d}$$

$$x_i, \lambda_i \geq 0 \qquad \forall i \in J \tag{10e}$$

$$y_{i,\ell} \geq 0 \qquad \forall i, \ell \in J. \tag{10f}$$

The LP (10) has $\Theta(n^2)$ variables and $O(n^2)$ constraints.

Lemma 5.1. *A feasible solution of (10) with integer x, y variables corresponds to a feasible solution of the max-weight interval scheduling problem on a path instance that satisfies conditions C.1–C.3, and vice versa.*

Proof. Assume we have a feasible solution to (10) with integer x, y variables; suppose $x_i = 1$. The right-hand side of some constraint (10c) is zero if $E(i)$ contains any pair of consecutive jobs, so we may assume that all minimal even pairs $(j, k) \in E(i)$ have $k - j \geq 3$. Furthermore, these constraints require $y_{i,\ell} = 1$ for some $\ell \in (j, k)$; we define $S_3 = \{\ell \in J : y_{i,\ell} = 1\}$. By construction, S_3 is a feasible schedule, as only one $y_{i,\ell}$ variable equals 1 in each minimal even pair. To define S_1 and S_2, we begin by adding to S_1 all jobs $j \in J$ with $w_j > w_i$; the remaining jobs are placed into

S_1 and S_2 in alternating fashion: For each minimal even pair $(j,k) \in E(i)$ with $\ell \in S_3 \cap (j,k)$, jobs $m \in (j,\ell)$ go into S_1 if they have j's parity and into S_2 otherwise. Similarly, jobs $m \in (\ell,k)$ go into S_1 if they have k's parity and into S_2 otherwise. The operation is analogous but simpler for "odd pairs," where we simply place jobs into S_1 and S_2 in alternating fashion. This construction ensures both S_1 and S_2 are feasible, as we never add adjacent jobs to the same schedule. This solution satisfies C.1, C.2, and C.3.

Now assume we have schedules S_1, S_2, S_3 satisfying C.1–C.3. To construct an integer solution to LP (10), take $i \in \operatorname{argmax}_{k \in S_2} w_k$ and set $x_i = 1$, with $x_j = 0$ otherwise. Condition C.3 requires that S_3 contain one job $\ell \in (j,k)$ for each $(j,k) \in E(i)$. Set the corresponding $y_{i,\ell} = 1$ and the remaining y variables to zero. Lastly, set $\lambda_i = \max_{\ell \in S_3} w_\ell$ and other λ variables to zero. Constraints (10c) are satisfied, as there is exactly one $y_{i,\ell}$ variable equal to one for each minimal pair in $E(i)$. Constraints (10d) are satisfied by construction. If $S_3 = \emptyset$, $E(i) = \emptyset$ as well, so (10c), (10d) are satisfied trivially. □

We can now establish that (10) solves the scheduling problem.

Proposition 5.2. *An optimal extreme point solution of* (10) *has integer x, y variables. Therefore,* (10) *solves the max-weight interval scheduling problem on path instances.*

Proof. We begin by analyzing the λ variables. At optimality, for each $i \in J$ some constraint (10d) must be tight to minimize each λ_i; thus

$$\lambda_i = \max_{(j,k) \in E(i)} \sum_{\ell \in (j,k)} w_\ell y_{i,\ell}.$$

For the maximizing interval (j,k) in this expression, (10c) requires $x_i = \sum_{\ell \in (j,k)} y_{i,\ell}$, so we minimize λ_i by setting $y_{i,m} = x_i$, where $m \in \operatorname{argmin}_{(j,k)} w_\ell$. Therefore, at optimality we have

$$\lambda_i = x_i \max_{(j,k) \in E(i)} \min_{j<\ell<k} w_\ell, \qquad i \in J,$$

where $\lambda_i = 0$ if $E(i) = \emptyset$. In particular, for optimal solutions λ is a linear function of the x variables.

By setting $c_i = \max_{(j,k) \in E(i)} \min_{j<\ell<k} w_\ell$, we project out y and λ so that (10) is equivalent to

$$\min\left\{ \sum_{i \in J} (w_i + c_i) x_i : \sum_{i \in J} x_i = 1; x \geq 0 \right\},$$

the optimization of a linear function over a simplex, which has integral extreme points. At optimality, $x_i = 1$ for some $i \in J$ and $x_j = 0$ otherwise. It follows from our reasoning above that we can take the y variables to be binary as well. The resulting optimal objective value corresponds to $f(S_2) + f(S_3)$ for an optimal set of schedules; recall that $f(S_1) = \max_{k \in J} w_k$ is a constant. □

6 Computational Study

In this section, we computationally evaluate our proposed methodology. Specifically, we examine the lower bound provided by the linear relaxation of (3), the upper bound given by our heuristics, and the performance of our branch-and-price algorithm, including its ability to improve both lower and upper bounds and to prove optimality.

In terms of objective functions, we use the max-weight cost as well as the square root, $f(S) = \sqrt{\sum_{i \in S} w_i}$, as a representative of non-decreasing, non-negative concave functions of total weight. We test our methods on both synthetic instances and instances derived from cloud computing usage data, and when possible also benchmark against the assignment formulation (2). We conduct all experiments on a MacBook with a 2.7GHz Dual-Core Intel i5 processor. Our base code uses Python 3.7.3, and the LP and IP solves use Gurobi 9.1 with default parameters unless otherwise noted. In the following sub-sections we discuss details of our implementation and instance design before summarizing our study's results.

6.1 Implementation Details

While a basic column generation algorithm can solve the linear relaxation of (3), in practice it is frequently enhanced with acceleration techniques, since the basic algorithm can suffer from slow convergence and other issues related to degeneracy. In our implementation, we use *in-out* column generation [6], which attempts to accelerate convergence by better selecting dual variables to use in the pricing problem. We include here a brief description of our implementation of the method, and refer the interested reader to [6] for more details: We maintain a dual feasible solution π_{in}, initialized to the zero vector, in addition to the potentially infeasible dual solution π_{out} obtained from the current solution to the restricted LP (4). We solve the pricing problem on the dual solution $\hat{\pi} = \gamma\pi_{\text{out}} + (1-\gamma)\pi_{\text{in}}$, a convex combination of the "in" and "out" solutions using convex multiplier $\gamma \in (0, 1]$. If $\hat{\pi}$ is infeasible, we generate a new column; otherwise, we update $\pi_{\text{in}} \leftarrow \hat{\pi}$. In addition to improving convergence times, the in-out method provides a way of obtaining valid dual bounds if the column generation algorithm does not converge within a time limit. Based on initial tests, we set $\gamma = 0.2$ at the root node and $\gamma = 1$ afterwards, equivalent to running the standard column generation algorithm after the root node. We observed that the in-out acceleration method significantly reduced solve time at the root node, but was slightly slower than the basic algorithm for subsequent nodes; the latter could be solved in a small number of iterations and suffered from fewer computational difficulties compared to the root node, likely because of the additional constraints.

We use the faster pricing algorithms at the root node and then the slower, more general algorithms after adding branching constraints. Specifically, for the max-weight function we solve the root LP using the LP-based pricing method and solve the remaining LP's with DP (7). For concave functions, we use the faster DP (9) to solve the root LP. For both functions, we perform full pricing and incorporate all improving columns after each pricing iteration.

There are various strategies to explore the branch-and-bound tree. In our implementation, at each node we branch on the most fractional $\theta_{i,j}$ value, the one closest to 0.5, breaking ties at random; if there is no fractional $\theta_{i,j}$, the solution is feasible and we do not branch. We explore the tree using an *iterative depth-first search* and process the 1-branch first, i.e. the branch with $\theta_{i,j} = 1$. The iterative depth-first search rule takes integer parameters α, β and follows standard depth-first search for the first α nodes. After we open α nodes, we restart the search from a random unprocessed node, reset the node count to zero, and update the threshold to $\alpha + \beta$. This process repeats until the search terminates. We designed our branching scheme in this manner after initial tests suggested that the lower bound obtained at the root node is often tight, and that the optimality gap tends to stem from the primal side. By biasing the search towards the 1-branches and performing iterative depth-first "dives", we attempt to steer the search to quickly obtain good feasible solutions while avoiding unpromising areas of the search space. In our experiments, we set the search parameters to $\alpha = 20$ and $\beta = 10$.

Finally, our branch-and-price algorithm obtains upper bounds in two ways. First, at the root node we obtain a feasible solution using the greedy and local-search heuristics described in Section 4.4; additionally, we use the schedules generated by these heuristics as the initial columns in the restricted LP (4) at the root node. At all other nodes, we implement a rounding heuristic; this heuristic obtains a feasible solution by greedily rounding the node's LP solution into a feasible solution for (3). We sort the schedules corresponding to non-zero variables in the LP solution by cumulative schedule weight. We add the schedule with highest cumulative weight to the solution, then re-sort the remaining schedules considering only the weights of jobs not covered by a previously included schedule. This process repeats until we obtain a feasible solution.

6.2 Instance Design

We use two types of instances to test our methods, randomly generated synthetic instances and instances constructed from a public cloud computing data set. We generate a random n-job instance by first specifying $t_{\max}$, the latest point at which jobs may arrive; each job i is created by sampling a start time s_i from the integer uniform distribution over interval $[0, t_{\max}]$ and sampling its length from the integer uniform distribution over $[1, 10]$.

For cloud instances, we use a publicly available Microsoft Azure data set [25]. The data set is comprised of two weeks of requests for virtual machine allocations; each request is associated with an arrival time and a departure time. The data set includes other application-specific data related to resource consumption; however, for our model we are only concerned with request start and end times. To generate an n-job instance, we select a random starting point in the data set, and take the next n arriving requests, creating jobs using their start and end times. We also check if the instance is sufficiently sparse to avoid easy cases in which feasible schedules contain only a few jobs. Specifically, we compute $\max_{C_i \in \mathbb{C}} |C_i|/n$, where $\max_{C_i \in \mathbb{C}} |C_i|$ is the cardinality of the largest clique in the conflict graph, and reject the instance when this ratio exceeds 0.3.

For both instance types we assign weights as uniform random integers in the interval $[1, 100]$.

We also considered generating weights correlated to job length, but found in initial tests that uncorrelated weights lead to more difficult instances. We generate *small* instances with $n = 100$, *moderate* instances with $n = 250$, and *large* instances with $n = 400$. In addition, for max-weight synthetic experiments we also test *very large* instances with $n = 550$.

6.3 Max-Weight Function: Small and Moderate Synthetic Instances

For small and moderate instances with $n \in \{100, 250\}$, we test our branch-and-price algorithm and also benchmark it against the assignment-type formulation (2). We use $t_{\max} \in \{n, n/2, n/5\}$, and generate ten instances for each pair $(n, t_{\max})$. We test both methods with a time limit of three hours; the results are summarized in Table 1. For each set of instances, we compare the geometric mean of the optimality gaps at termination, the average running time of instances where the method proves optimality (in seconds), and the percentage of the instances that are solved to optimality. We also compute the geometric means of the optimality gaps at the root node.

Table 1: Branch-and-price and assignment formulation experiments with the max-weight function on small, moderate synthetic instances.

n	$t_{\max}$	Root Gap	B&P Gap	Sol. Time	% Sol.	Assign. Gap	Sol. Time	% Sol.
100	100	0.94%	0.00%	41.56	100%	0.00%	436.45	100%
100	50	3.25%	0.00%	43.04	100%	0.75%	1199.34	60%
100	20	1.66%	0.00%	15.01	100%	0.68%	405.86	30%
250	250	1.54%	0.00%	5892.44	100%	2.01%	2228.56	50%
250	125	2.64%	0.00%	2779.20	100%	3.02%	-	0%
250	50	3.11%	0.00%	1006.74	100%	4.46%	-	0%

The results verify that the proposed branch-and-price algorithm outperforms the assignment formulation. For every test instance, the branch-and-price algorithm is able to prove optimality within three hours. The class $(n, t_{\max}) = (100, 100)$ is the only one where the assignment benchmark solves all instances to optimality in the same time. Conversely, the assignment formulation did not prove optimality for any of the $(250, 125)$ or $(250, 50)$ instances. The average solution times are lower for the branch-and-price algorithm, and it scales better as the instances become denser, while the assignment formulation struggles with dense instances. As a final observation, the root LP's lower bound in the branch-and-price tree is often tight, with any gap stemming from the primal side. In contrast, with the assignment formulation the IP solver is often able to find a strong primal solution, but struggles to improve the lower bound.

6.4 Max-Weight Function: Large and Very Large Synthetic Instances

To assess the scalability of our branch-and-price algorithm for the max-weight function, we also considered larger synthetic instances. We use $n \in \{400, 550\}$ and $t_{\max} = n/5$. The choice of $t_{\max}$

is motivated by the prior experiments, which indicate that denser instances have larger root gaps but also that the algorithm may scale well. For each pair n, we generate ten random instances and use a six-hour time limit; Table 2 summarizes the results. We report the running times and the optimality gaps at termination and at the root node. We also report the geometric means of final optimality gaps and the average solution time of instances that are solved to optimality.

Table 2: Branch-and-price experiments with the max-weight function on large and very large synthetic instances.

n	$t_{\max}$	Root Gap	B&P Gap	Sol. Time	n	$t_{\max}$	Root Gap	B&P Gap	Sol. Time
400	80	0.52%	0.00%	8697.42	550	110	3.68%	3.68%	TL
400	80	4.76%	0.00%	10468.31	550	110	3.12%	3.12%	TL
400	80	3.92%	0.00%	9235.76	550	110	5.39%	5.39%	TL
400	80	1.33%	0.00%	7371.16	550	110	2.94%	2.94%	TL
400	80	3.16%	0.00%	9328.55	550	110	2.31%	2.31%	TL
400	80	1.04%	0.00%	9642.15	550	110	0.95%	0.95%	TL
400	80	1.12%	0.00%	8437.06	550	110	3.16%	3.16%	TL
400	80	2.36%	0.00%	9890.78	550	110	2.76%	2.76%	TL
400	80	3.00%	0.00%	9531.93	550	110	2.86%	2.86%	TL
400	80	2.05%	0.00%	7893.05	550	110	4.54%	4.54%	TL
	Avg.	**1.92%**	**0.00%**	**9049.62**		**Avg.**	**2.92%**	**2.92%**	

The results indicate that the branch-and-price algorithm scales well to large instances, and is able to reliably prove optimality in less than six hours. For the largest instances, the algorithm is unable to reduce the root optimality gap. This difficulty stems from at least two sources: First, the larger job number necessitates a much larger tree to improve the bound and prove optimality. Second, and equally important, the LP solves at each node take significantly longer, meaning we can process a smaller number of nodes within the time limit. Nonetheless, already at the root node we obtain solutions and bounds yielding an average optimality gap of less than 3%. Based on previous experiments, we conjecture that this gap stems primarily from the primal side; improved heuristics could potentially help prove optimality.

6.5 Square Root Function: Synthetic Instances

For the square root function we limit ourselves to the root node; equivalently, we solve the linear relaxation of (3) and run primal heuristics. We conduct our experiment in this manner for two reasons: Our initial tests suggested that this approach often suffices to prove optimality for small and moderate instances, while for large instances the LP relaxation itself takes a significant amount of time. For each combination of $n \in \{100, 250, 400\}$ and $t_{\max} = n/5$, we generate ten instances, and run our test with a six-hour time limit; Table 3 summarizes the results. For each instance class, we report the geometric mean of the optimality gaps, the average running time for instances

where the column generation algorithm converges, the percentage of instances where the column generation algorithm converges, and the percentage of instances where we prove optimality.

Table 3: Column generation experiments with the square root function on synthetic instances.

n	$t_{\max}$	CG Gap	CG Time	% Conv.	% Sol.
100	20	0.00%	252.55	100%	100%
250	50	0.30%	4959.18	70%	70%
400	80	1.30%	-	0%	0%

The results indicate that we can prove optimality for all instances where the column generation algorithm converges. The column generation does not converge within the time limit for 30% of the moderate instances and all of the large instances, and we use the dual bounds provided by the in-out acceleration scheme. However, even for the large instances the average optimality gap is only 1.3%. Compared with the max-weight function, the pseudo-polynomial complexity of the pricing algorithm for the square root function makes solving the LP more computationally difficult, but the resulting lower bound is typically stronger.

6.6 Cloud Data Instances

As a final experiment, we test our methods on instances derived from cloud computing data, as described in Section 6.2. For the max-weight function, we test instances of all sizes, with $n \in \{100, 250, 400, 550\}$, and for the square root function we exclude very large instances and use $n \in \{100, 250, 400\}$; for each function and each n we generate ten instances. For the max-weight function, we test the branch-and-price algorithm, and for the square root function we solve the root node's LP relaxation.

Table 4 summarizes the results for the max-weight function. We report the geometric means of the root and terminal optimality gaps, the average running times of solved instances, and the percentage of instances solved.

Table 4: Branch-and-price experiments with the max-weight function on cloud instances.

n	Root Gap	B&P Gap	Sol. Time	% Sol.
100	0.32%	0.00%	16.15	100%
250	0.53%	0.00%	864.48	100%
400	0.94%	0.15%	8589.58	80%
550	0.81%	0.81%	5584.67	40%

In general, we see that the application instances are easier than the synthetic instances. For example, the branch-and-price algorithm is able to solve 40% of the very large instances with the max-weight function, versus none of the analogous synthetic instances. Furthermore, the root

optimality gaps are much lower than for the synthetic instances, less than 1% in all cases. An explanation for this behavior may be the density, which tends to be higher in the cloud data; correspondingly, the average solution times are similar to those of the synthetic instances with $t_{\max} = n/5$.

Table 5 summarizes the results for the square root function. We report average running times of instances where the column generation algorithm converges, the geometric mean of the optimality gaps at termination, the percentage of instances for which the algorithm converges, and the percentage of instances solved.

Table 5: Column generation experiments with the square root function on cloud instances.

n	CG Gap	CG Time	% Conv.	% Sol.
100	0.01%	453.52	100%	90%
250	0.23%	6990.59	90%	60%
400	1.75%	17791.94	10%	0%

Overall, the column generation algorithm converges for more instances than the corresponding synthetic experiment, but this does not lead to a significant change in average optimality gaps. In contrast to the synthetic instances, the column generation algorithm may converge without proving optimality. For example, despite converging for all moderate instances but one, the resulting gaps are slightly lower than for the corresponding synthetic instances.

7 Conclusions

In this work, we propose an exact optimization approach for two classes of interval scheduling problems exhibiting economies of scale; to our knowledge, this is the first such approach proposed for these problems. Our approach is based on column generation and a set covering formulation, using feasible schedules as decision variables. For the max-weight function and functions of cumulative weight, we describe how to solve the linear relaxation of this formulation and provide efficient pricing algorithms. To obtain integer solutions, we extend this method to a full branch-and-price algorithm and provide a detailed account of the relevant design decisions. As a secondary result, we also provide a compact integral formulation for the max-weight function on path instances, a polynomially solvable case for which no such formulation was known. Our computational study provides evidence of our proposed method's effectiveness. We can provably optimize instances with as many as 400 jobs and can otherwise give solutions and bounds with very small gaps.

While we have demonstrated the effectiveness of the set covering formulation on our chosen objective functions, future work can extend the approach to other classes of functions. The major requirement is to determine when the pricing problem can be solved efficiently. Both of the problems studied belong to the more general class of submodular interval scheduling problems.

In this case, the general form of the pricing problem is supermodular maximization subject to interval packing constraints. To our knowledge, this specific problem has not been previously addressed in the literature.

Another area of future work concerns how to leverage polyhedral results for the column generation model. In our computational study, we observed that the root LP often provides a tight lower bound; however, it is easy to construct instances in which this is not the case, even when the conflict graph is a path. For these instances, the lower bound could possibly be strengthened by the addition of valid inequalities.

Acknowledgements

The authors' work was partially supported by the U.S. National Science Foundation via grant CMMI-1552479. Christopher Muir's work was also supported via a U.S. NSF Graduate Research Fellowship.

References

[1] Enrico Angelelli, Nicola Bianchessi, and Carlo Filippi. Optimal interval scheduling with a resource constraint. *Computers & operations research*, 51:268–281, 2014.

[2] Esther M Arkin and Ellen B Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1):1–8, 1987.

[3] Nurşen Aydın, İbrahim Muter, and Ş. İlker Birbil. Multi-objective temporal bin packing problem: An application in cloud computing. *Computers & Operations Research*, 121:104959, 2020.

[4] Reuven Bar-Yehuda, Magnús M Halldórsson, Joseph Naor, Hadas Shachnai, and Irina Shapira. Scheduling split intervals. *SIAM Journal on Computing*, 36(1):1–15, 2006.

[5] Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

[6] Walid Ben-Ameur and José Neto. Acceleration of cutting-plane and column generation algorithms: Applications to network design. *Networks: An International Journal*, 49(1):3–17, 2007.

[7] Peter Brucker and Lars Nordmann. The k-track assignment problem. *Computing*, 52(2):97–122, 1994.

[8] Jean Cardinal, Samuel Fiorini, and Gwenaël Joret. Minimum entropy coloring. In *International Symposium on Algorithms and Computation*, pages 819–828. Springer, 2005.

[9] Martin C Carlisle and Errol L Lloyd. On the k-coloring of intervals. *Discrete Applied Mathematics*, 59(3):225–235, 1995.

[10] José R Correa and Nicole Megow. Clique partitioning with value-monotone submodular cost. *Discrete Optimization*, 15:26–36, 2015.

[11] Diego Delle Donne, Fabio Furini, Enrico Malaguti, and Roberto Wolfler Calvo. A branch-and-price algorithm for the minimum sum coloring problem. *Discrete Applied Mathematics*, 2020.

[12] Mauro Dell'Amico, Fabio Furini, and Manuel Iori. A branch-and-price algorithm for the temporal bin packing problem. *Computers & Operations Research*, 114:104825, 2020.

[13] Guy Desaulniers, Jacques Desrosiers, and Marius M Solomon. *Column generation*, volume 5. Springer Science & Business Media, 2006.

[14] Bruno Escoffier, Jérôme Monnot, and Vangelis Th. Paschos. Weighted coloring: further complexity and approximability results. *Information Processing Letters*, 97(3):98–103, 2006.

[15] Takuro Fukunaga, Magnús M Halldórsson, and Hiroshi Nagamochi. "Rent-or-buy" scheduling and cost coloring problems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 84–95. Springer, 2007.

[16] Takuro Fukunaga, Magnús M Halldórsson, and Hiroshi Nagamochi. Robust cost colorings. In *SODA*, volume 8, pages 1204–1212, 2008.

[17] Fabio Furini and Enrico Malaguti. Exact weighted vertex coloring via branch-and-price. *Discrete Optimization*, 9(2):130–136, 2012.

[18] Fabio Furini, Enrico Malaguti, and Alberto Santini. An exact algorithm for the partition coloring problem. *Computers & Operations Research*, 92:170–181, 2018.

[19] Fabio Furini and Xueying Shen. Matheuristics for the temporal bin packing problem. In Lionel Amodeo, El-Ghazali Talbi, and Farouk Yalaoui, editors, *Recent Developments in Metaheuristics*, pages 333–345. Springer International Publishing, Cham, 2018.

[20] Virginie Gabrel. Scheduling jobs within time windows on identical parallel machines: New model and algorithms. *European Journal of Operational Research*, 83(2):320–329, 1995.

[21] Dion Gijswijt, Vincent Jost, and Maurice Queyranne. Clique partitioning of interval graphs with submodular costs on the cliques. *RAIRO-Operations Research*, 41(3):275–287, 2007.

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

[23] Gupta, Lee, and Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, C-28(11):807–810, 1979.

[24] Udaiprakash I Gupta, Der-Tsai Lee, and JY-T Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467, 1982.

[25] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:{VM} allocation service at scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 845–861, 2020.

[26] Xiaoyun Ji and John E Mitchell. Branch-and-price-and-cut on the clique partitioning problem with minimum clique size requirement. *Discrete Optimization*, 4(1):87–102, 2007.

[27] Mikhail Y. Kovalyov, C.T. Ng, and T.C. Edwin Cheng. Fixed interval scheduling: Models, applications, computational complexity and algorithms. *European Journal of Operational Research*, 178(2):331–342, 2007.

[28] Soonhui Lee, Jonathan Turner, Mark S. Daskin, Tito Homem de Mello, and Karen Smilowitz. Improving fleet utilization for carriers by interval scheduling. *European Journal of Operational Research*, 218(1):261–269, 2012.

[29] Silvano Martello and Paolo Toth. A heuristic approach to the bus driver scheduling problem. *European Journal of Operational Research*, 24(1):106–117, 1986. OR and Microcomputers Miscellaneous OR Applications.

[30] Anuj Mehrotra and Michael A Trick. A column generation approach for graph coloring. *informs Journal on Computing*, 8(4):344–354, 1996.

[31] Chi To Ng, Tai Chiu Edwin Cheng, Andrei M Bandalouski, Mikhail Y Kovalyov, and Sze Sing Lam. A graph-theoretic approach to interval scheduling on dedicated unrelated parallel machines. *Journal of the Operational Research Society*, 65(10):1571–1579, 2014.

[32] Stephan Olariu. An optimal greedy heuristic to color interval graphs. *Information Processing Letters*, 37(1):21–25, 1991.

[33] Siwate Rojanasoonthon, Jonathan F Bard, and Surender D Reddy. Algorithms for parallel machine scheduling: a case study of the tracking and data relay satellite system. *Journal of the Operational Research Society*, 54(8):806–821, 2003.

[34] Frits CR Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2(5):215–227, 1999.