ORIGINAL ARTICLE

Journal Section

An exact bidirectional pulse algorithm for the constrained shortest path

Nicolás Cabrera[1] | Andrés L. Medaglia[1] | Leonardo Lozano[2] | Daniel Duque[3]

[1]Industrial Engineering Department, Universidad de los Andes, Carrera 1 Este No. 19 A - 40, 111711, Bogotá, Colombia

[2]Operations, Business Analytics & Information Systems, University of Cincinnati, 2906 Woodside Drive, Cincinnati, Ohio, U.S.

[3]Industrial Engineering and Management Sciences, Northwestern University, 2145 Sheridan Rd, Evanston, Illinois, U.S.

Correspondence
Andrés L. Medaglia PhD, Industrial Engineering Department, Universidad de los Andes, Carrera 1 Este No. 19 A - 40, 111711, Bogotá, Colombia
Email: amedagli@uniandes.edu.co

Funding information
Office of Naval Research under Grant N00014-19-1-2329

A constrained shortest path is a minimum-cost sequence of arcs on a directed network that satisfies knapsack-type constraints on the resource consumption over the arcs. We propose an exact method based on a recursive depth-first search procedure known as the pulse algorithm. One of the key contributions of the proposal lies in a bidirectional search strategy leveraged on parallelism. In addition, we developed a pulse-based heuristic that quickly finds near-optimal solutions and shows great potential for column generation schemes. We present computational experiments over large real-road networks with up to 6 million nodes and 15 million arcs. We illustrate the use of the bidirectional pulse algorithm in a column generation scheme to solve a multi-activity shift scheduling problem, where the pricing problem is modeled as a constrained-shortest path with multiple resource constraints.

KEYWORDS
constrained shortest path, bidirectional search, large-scale networks

1 | INTRODUCTION

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be a directed graph defined by a set of nodes $\mathcal{N} = \{v_1, \ldots, v_n\}$ and a set of directed arcs $\mathcal{A} = \{(i,j) | v_i \in \mathcal{N}, v_j \in \mathcal{N}, i \neq j\}$. Each arc $(i,j) \in \mathcal{A}$ has a corresponding nonnegative cost c_{ij} and a nonnegative resource consumption vector $\mathbf{t}_{ij} \in \mathbb{R}^m$. The constrained shortest path problem (CSP) consists on finding a path $\mathcal{P}$ between a start node $v_s \in \mathcal{N}$

and an end node $v_e \in \mathcal{N}$ that minimizes the total cost, without exceeding a maximum resource consumption vector $\mathbf{T} \in \mathbb{R}^m$. Unless stated otherwise, for simplicity in the exposition, we assume throughout the paper that $m = 1$, thus the resource consumption t_{ij} and resource limit T are scalars. The CSP is known to be NP-Hard even for the case of one resource [1].

The CSP naturally arises in a wide range of transportation and logistics applications. Zabarankin et al. [2] model an aircraft's flight trajectory problem in which the goal is to minimize exposure to radars while satisfying technical constraints such as trajectory length. Cabral et al. [3] model a telecommunication network design in which the goal is to find a set of edges that allow for minimum cost paths between communicating pairs of nodes while satisfying a constraint on the path length (in the case of a single pair of communicating nodes, the problem reduces to the CSP). In addition, the CSP is often used as an *auxiliary problem* in column generation (CG) schemes. CG is a technique that solves linear programs with a large number of decision variables. Instead of considering all the variables at once, CG starts with a modestly-sized subset of variables and incorporates candidate variables in an iterative fashion. At every iteration of a CG scheme, the goal is to either prove optimality or choose a promising decision variable to enter the basis. This goal is accomplished by solving an optimization problem, known as the auxiliary problem [4]. Examples of problems in which variants of the CSP appear as an auxiliary problem in a CG scheme include planning and routing [5], flight planning [6], crew pairing [7], shift scheduling [8], and tail assignment in aircraft scheduling [9], among others.

There is significant literature addressing the CSP through different solution strategies including dynamic programming-based (DP) labeling algorithms [10, 11], Lagrangian relaxation [12], and path ranking approaches [13, 14, 15] among others. DP-based algorithms have the distinctive feature that they save the state of the search using labels. Although they could be extremely fast, they might fail to scale well in very large instances due to the well-known "curse of dimensionality". Methods based on Lagrangian relaxation take advantage of the effectiveness of methods to solve the unconstrained shortest path, by relaxing the side resource constraints. Ranking approaches solve the CSP by using a k-th shortest path algorithm that identifies multiple paths that are later sorted and evaluated.

We highlight two works from the literature that are closely related to our approach. The first one is the pulse algorithm by Lozano and Medaglia [16]. This method is based on a recursive depth-first search that combines various pruning strategies to avoid complete exploration of the solution space. One key aspect of the pulse algorithm is that it uses a limited number of labels to discard dominated partial paths; however, these labels are never extended, nor the exactness of the algorithm depends on storing the complete set of nondominated labels at each node. We also focus our attention on the recent bidirectional A* algorithm by Thomas et al. [17]. Their labeling approach searches for paths in the network from both the starting and ending node, storing the complete set of nondominated labels for each node in both directions until reaching a stopping criterion. Complete paths are then obtained by merging labels coming from opposite directions. To avoid the curse of dimensionality associated with labeling algorithms, the authors use a set of pruning strategies to avoid extending suboptimal or infeasible labels. Computationally, the bidirectional A* algorithm outperforms the pulse algorithm and compares favorably with the best-known methods for CSP on a set of very large instances. As a result, we consider bidirectional A* to be one of the current state-of-the-art algorithms in the literature and use it as a benchmark for our computations.

Historically, some bidirectional label-setting methods have suffered from poor computational performance. Pohl [18] argued that the reason for this is that paths starting in one direction do not meet with those coming from the other direction. This notion is commonly known as the "crossing missiles metaphor". Most of the solutions to solve this problem focus on joining labels created from both search directions and defining a search perimeter. For example, Thomas et al. [17] defined a perimeter based on the resource consumption. Specifically, if a label had consumed more than half of the resource, the label is not expanded. The definition of this perimeter implies storing all non-dominated labels which may lead to scalability problems. In contrast, we do not define a search perimeter nor the correctness of our algorithm depends on storing all non-dominated labels. However, we do consider a strategy to link partial paths created in both search directions using a limited number of labels.

The pulse algorithm has been successfully extended for the elementary shortest path problem with resource constraints [19], the biobjective shortest path problem [20], the weight constrained shortest path problem with replenishment [21], the orienteering problem with time windows [22], and more recently, the robust shortest path problem [23]. Beyond the domain of shortest path problems, several authors have used the pulse algorithm as a component to solve other hard combinatorial problems. For instance, the pulse algorithm has been used in network interdiction [24], shift scheduling [8], evasive flow [25], resource constrained pickup-and-delivery [26], and green vehicle routing problems [27], among others.

The contribution of this paper is twofold. From a methodological perspective, we propose an alternative approach to conduct bidirectional search that is not based on extending labels. Our approach builds on the ideas from the pulse algorithm by performing an exact bidirectional adjustable depth-first (or breadth-first) search executed in parallel both from the starting and ending node. We use a limited number of labels per node to enforce dominance relationships and propose additional strategies based on completing and merging partial paths represented by these labels. Since we enforce a strict limit on the number of labels stored at each node, we avoid the curse of dimensionality while still ensuring optimality as the exactness of our algorithm does not depend on storing a complete set of nondominated labels. From a computational perspective, we compare favorably with the state-of-the-art algorithm by Thomas et al. [17] over a set of 360 instances from real road networks in the US commonly used in the literature. Additionally, we embedded our algorithm in a column generation scheme to solve the linear relaxation of the multi-activity shift scheduling problem (MASSP), which involves solving a CSP with multiple resources. Moreover, we present a pulse-based heuristic that provides high-quality feasible solutions fast.

This paper is organized as follows. Section 2 presents an overview of the proposed algorithm. Section 3 provides a detailed description of the proposed acceleration strategies. Section 4 presents the computational experiments. Section 5 provides a sensitivity analysis on the algorithm's components. Section 6 presents our pulse-based heuristic for the CSP. Finally, Section 7 concludes the paper and outlines future work.

2 | BIDIRECTIONAL PULSE ALGORITHM: INTUITION AND OVERVIEW

In this section, we provide a high-level explanation of our proposed algorithm. For completeness, Section 2.1 presents a summary of the original pulse algorithm. Section 2.2 provides an overview of our proposed bidirectional search while Section 2.3 describes the pseudocode of our algorithm and highlights the proposed acceleration strategies.

2.1 | Pulse algorithm

The original pulse algorithm (PA) by Lozano and Medaglia [16] is a recursive search based on the idea of propagating a *pulse* through the network, starting at node v_s. Pulses represent partial paths and propagate through the outgoing arcs from each node, storing crucial information about the partial path being explored. Once a pulse corresponding to a feasible path reaches the end node v_e, the PA tries to update an upper (or primal) bound on the objective function, stops the pulse propagation, and backtracks to continue the recursive search resulting in a pure depth-first exploration. Opposite to labels, pulses are not stored in memory but passed as arguments in the recursive search function. If nothing prevents pulses from propagating, the PA enumerates all feasible paths from v_s to v_e ensuring that an optimal path $\mathcal{P}^*$ is found.

To avoid a complete enumeration of the solution space, the PA relies on a set of pruning strategies proposed to rapidly and effectively prune pulses, without cutting off the optimal solution. The core strategies in the PA are pruning by infeasibility, bounds, and dominance. The infeasibility pruning strategy (see §3.1.1) uses a lower bound on the minimum resource required to reach the end node from any given intermediate node and discards partial paths that cannot be completed into a feasible v_s to v_e path. The bounds pruning strategy (see §3.1.2) uses lower bounds on the minimum-cost paths from any intermediate node to the end node and discards partial paths that cannot be part of an optimal solution. The dominance pruning strategy (see §3.1.3) uses a limited number of labels to store information on partial paths explored during the search and discard partial paths based on traditional dominance relationships. In contrast to labeling algorithms, the PA does not rely on extending every non-dominated label for correctness; instead, this is guaranteed by properly truncating the recursive search. Hence, even if no labels were used at any of the nodes, the algorithm remains correct.

2.2 | Bidirectional recursive search

We propose a bidirectional pulse algorithm (BP) that explores the network in both search directions. The BP propagates pulses simultaneously from the start node v_s to the end node v_e, resulting in a *forward* search (f); and from the end node v_e to the start node v_s, resulting in a *backward* search (b). Regardless of the search direction, each pulse traverses the network building a partial path $\mathcal{P}$, while storing the cumulative cost $c(\mathcal{P})$ and the cumulative resource consumption $t(\mathcal{P})$. For the sake of clarity in the exposition, we denote a partial path $\mathcal{P}$ arriving to node v_i by $\mathcal{P}(i)$ and a partial path P coming from v_j to v_i by $\mathcal{P}(j,i)$, where partial paths of the form $\mathcal{P}(s,i)$ and $\mathcal{P}(j,e)$ correspond to the forward search while paths of the form $\mathcal{P}(e,i)$ and $\mathcal{P}(j,s)$ correspond to the backward search. Henceforth, we use $\mathcal{P}$, $\mathcal{P}(i)$, and $\mathcal{P}(j,i)$ interchangeably depending on the context. We also refer to an initial node v_α and a destination node v_ω for both search directions, where $v_\alpha, v_\omega \in \{v_s, v_e\}$ and, with a slight abuse of notation, we avoid the use of sub- or superscripts to indicate the search direction.

In the absence of pruning strategies, the BP enumerates all the paths in the network twice, once in the forward search and once in the backward search. Although both single-thread and parallel implementations of the BP are conceivable, we proceed in our explanation under a parallel mindset as it is more natural to understand (and implement) the algorithm[1]. We propose a bidirectional search paradigm that leverages the fact that both the forward and the backward search are executed concurrently to update the primal bound faster compared to a single-directional search, while using additional pruning strategies based on the partial paths being explored in both directions. We include all the core pruning strategies from the PA in both directions and propose two additional strategies, namely, *path completion* and *path joins*. The intuition behind these primal bound-update strategies is to construct feasible paths before reaching the destination node on either the forward or the backward direction.

A fundamental difference between the BP and traditional bidirectional algorithms is that BP does not rely on labels to guide the search and thus it does not require storing a possibly exponential number of nondominated labels for each search direction. However, we do use a limited number of labels for the dominance pruning and path joining strategies. At every node v_i, we store a list of non-dominated labels $\mathcal{L}^k(v_i) = \{(c_{il}, t_{il}) \mid l = 1, \ldots, R\}$ corresponding to partial paths created on the search direction $k \in \{f, b\}$, where R denotes the total number of labels (i.e., memory size), and c_{il} and t_{il} are the cost and resource consumption, respectively. It is important to note that the correctness of our algorithm does not rely on storing all non-dominated labels and choosing $R \ll \infty$ poses a tradeoff between the ability to efficiently prune pulses and the amount of memory required to execute the algorithm.

[1]In a single-thread implementation, the BP could alternate between the forward and the backward search taking turns, but a rule to switch between directions would be required.

In the BP, we also address a problem that stems from the pure depth-first search exploration done by the PA. On some networks, a pure depth-first search could explore vast unpromising regions of the search space, before backtracking and correcting poor decisions made earlier. To overcome this problem, we adapt the pulse queueing strategy proposed by Bolívar et al. [21] for the Weight Constrained Shortest Path Problem with Replenishment (WCSPP-R). Formally, let δ be a depth limit, and for each search direction k, let Q^k be a pulse queue that stores *halted* partial paths traveling on the search direction k. We describe the queueing procedure indistinctly of the search direction as both queues operate independently. A pulse is halted for the first time when it is extended δ additional nodes from the origin node. Upon dequeueing a pulse $\mathcal{P}$ from Q^k, the recursive search is resumed from the last node in $\mathcal{P}$. Halting a pulse entails saving the partial path $\mathcal{P}$ in the corresponding queue (along with the node where the pulse was paused $n(\mathcal{P})$, the cumulative cost $c(\mathcal{P})$, and the resource consumption $t(\mathcal{P})$) and backtracking to resume the current recursion stack. Note that when a pulse $\mathcal{P}$ is dequeued, the recursive search will never backtrack to the node visited before $n(\mathcal{P})$. Also note that a pulse is only dequeued when the partial path that is currently propagating is exhausted, i.e., when all its descendant paths halt, get pruned, or reach the end node. Pulses are dequeued following a queue discipline (i.e., a rule to select the next pulse to be dequeued) that we defer until Section 3.3. Finally, observe that for $\delta = 0$ the BP results in a pure breadth-first search, whereas for $\delta = \infty$ the BP reduces to a pure depth-first search.

Let $\mathcal{G}(k)$ be the directed network to be traversed on the search direction $k \in \{f, b\}$. Let the *forward* network $\mathcal{G}(f) = (\mathcal{N}, \mathcal{A}(f))$ be the original directed network $\mathcal{G} = (\mathcal{N}, \mathcal{A})$. Let $\mathcal{G}(b) = (\mathcal{N}, \mathcal{A}(b))$ denote the *backward* (i.e., reversed) network for which $\mathcal{A}(b) = \{(j, i) | (i, j) \in \mathcal{A}(f)\}$. The BP starts with an initialization step that computes minimal resource consumption paths $\mathcal{P}^t(i, s)$ and $\mathcal{P}^t(i, e)$, and minimal cost paths $\mathcal{P}^c(i, s)$ and $\mathcal{P}^c(i, e)$, from every node v_i to both the start node v_s and the end node v_e, using graphs $\mathcal{G}(f)$ and $\mathcal{G}(b)$, accordingly. These paths provide lower bounds on resource consumption and cost used in most of the pruning strategies. After the initialization step, the BP executes the `pulseSearch` function (described in Algorithm 1 below) initiating the recursive search on the forward direction by invoking `pulseSearch`$(\mathcal{G}(f), T, v_s, v_e, f)$ and in the backward direction by invoking `pulseSearch`$(\mathcal{G}(b), T, v_e, v_s, b)$ on separate concurrent threads. The BP stops as soon as one of the searches is completed.

2.3 | Bidirectional pulse algorithm

Algorithm 1 presents the main logic of the pulse search algorithm. Line 1 initializes the best incumbent path $\mathcal{P}^*$. Lines 2 and 3 initialize the pulse queue Q^k with a paused pulse at the initial node v_α. Lines 4-7 propagate pulses stored in Q^k. Line 5 selects the next pulse to be processed given the queue discipline and removes it from Q^k. Line 6 propagates the pulse (see Algorithm 2) resuming the exploration at node $n(\mathcal{P})$ and setting the path depth to zero. Finally, line 8 returns an optimal path $\mathcal{P}^*$, which is obtained as the best incumbent path through the recursion.

Algorithm 1 `pulseSearch` function

```
Require: $\mathcal{G}(k)$, directed graph; $T$, resource constraint; $v_\alpha$, initial node; $v_\omega$, destination node; $k$, search direction.
Ensure: $\mathcal{P}^*$, optimal path.
1: $\mathcal{P}^* \leftarrow \emptyset$
2: $\mathcal{P} \leftarrow \{v_\alpha\}$
3: push($\mathcal{Q}^k, \mathcal{P}$)
4: while $\mathcal{Q}^k \neq \emptyset$ do
5:     $\mathcal{P} \leftarrow$ pop($\mathcal{Q}^k$)                                  ▷ see §3.3
6:     pulse($n(\mathcal{P}), c(\mathcal{P}), t(\mathcal{P}), 0, \mathcal{P}, k$)    ▷ see Algorithm 2
7: end while
8: return $\mathcal{P}^*$
```

Algorithm 2 shows the body of the recursive pulse function. The set $\Gamma^+(v_i) = \{v_j \in \mathcal{N} \mid (i,j) \in \mathcal{A}(k)\}$ comprises the nodes directly connected to node v_i (on search direction k). Lines 2 through 5 update the cumulative cost, the resource consumption, the pulse depth, and the partial path. Lines 6-9 perform Boolean function checks that return `true` if the incoming pulse to node v_j is pruned (they return `false`, otherwise). Line 6 checks if it is feasible to reach the destination node v_ω from node v_j given that the path has already a resource consumption of t'. Line 7 tries to prune the pulse using lower bounds on the best cost achievable by the current partial path. Line 8 checks the dominance relations of $\mathcal{P}'$ against a list of non-dominated partial paths $\mathcal{L}^k(v_j)$ for node v_j. Line 9 explores the possibility of completing the partial path $\mathcal{P}'$ with the minimum cost feasible path. In case the path is completed, the path $\mathcal{P}^*$ is updated. Line 10 tries to join partial paths using the lists of non-dominated labels for node v_j that were created in the opposite direction. If a path is successfully joined, the path $\mathcal{P}^*$ is updated. Line 11 checks if the path has advanced δ additional steps, if so, line 12 adds it to the pulse queue $\mathcal{Q}^k$; otherwise, line 14 recursively propagates the pulse through node v_j.

Algorithm 2 `pulse` function

```
Require: $v_i$, node; $c$, cumulative cost; $t$ cumulative resource; $d$, depth; $\mathcal{P}$, partial path; $k$, search direction.
 1: for $v_j \in \Gamma^+(v_i)$ do
 2:     $c' \leftarrow c + c_{ij}$
 3:     $t' \leftarrow t + t_{ij}$
 4:     $d' \leftarrow d + 1$
 5:     $\mathcal{P}' \leftarrow \mathcal{P} \cup \{v_j\}$
 6:     if $\neg$check_feasibility($v_j, t', k$) then                          ▷ see §3.1.1
 7:         if $\neg$check_bounds($v_j, c', k$) then                           ▷ see §3.1.2
 8:             if $\neg$check_dominance($v_j, c', t', k$) then                ▷ see §3.1.3
 9:                 if $\neg$check_complete_path($v_j, c', t', k$) then        ▷ see §3.2.1
10:                     join_paths($v_j, c', t', k$)                           ▷ see §3.2.2
11:                     if $d' > \delta$ then
12:                         push($Q^k, \mathcal{P}'$)
13:                     else
14:                         pulse($v_j, c', t', d', \mathcal{P}', k$)
15:                     end if
16:                 end if
17:             end if
18:         end if
19:     end if
20: end for
21: return void
```

Figure 1 shows a schematic summary of the bidirectional pulse algorithm. Note that the BP constructs paths on the forward (blue) and the backward (green) direction. Every time a partial path $\mathcal{P}(i)$ reaches an intermediate node v_i, we check if it can be safely pruned using the core pruning strategies (i.e., infeasibility, bounds, and dominance). If this fails, we use the path completion strategy that attempts to construct a feasible path to the destination node v_ω. This strategy adds the minimum cost path $\mathcal{P}^c(i, \omega)$ and the minimum resource consumption path $\mathcal{P}^t(i, \omega)$. With the same purpose, the path joins strategy uses information stored on $\mathcal{L}^k(v_i)$ about partial paths created on the opposite search direction. Finally, we check if the partial path has reached the maximum depth limit. If so, the partial path is added to the corresponding pulse queue Q^k.

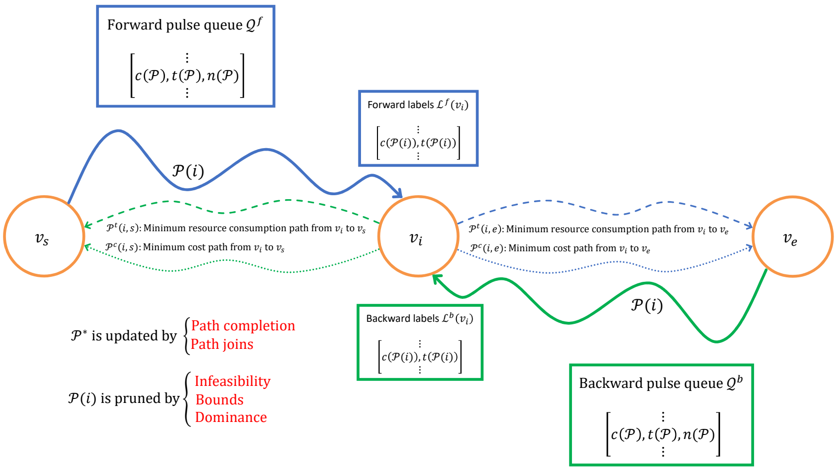

Forward pulse queue $\mathcal{Q}^f$
$[c(\mathcal{P}), t(\mathcal{P}), n(\mathcal{P})]$
Forward labels $\mathcal{L}^f(v_i)$
$[c(\mathcal{P}(i)), t(\mathcal{P}(i))]$
$\mathcal{P}(i)$
v_s
v_i
v_e
$\mathcal{P}^t(i,s)$: Minimum resource consumption path from v_i to v_s
$\mathcal{P}^c(i,s)$: Minimum cost path from v_i to v_s
$\mathcal{P}^t(i,e)$: Minimum resource consumption path from v_i to v_e
$\mathcal{P}^c(i,e)$: Minimum cost path from v_i to v_e
$\mathcal{P}^*$ is updated by Path completion, Path joins
$\mathcal{P}(i)$ is pruned by Infeasibility, Bounds, Dominance
Backward labels $\mathcal{L}^b(v_i)$
$[c(\mathcal{P}(i)), t(\mathcal{P}(i))]$
Backward pulse queue $\mathcal{Q}^b$
$[c(\mathcal{P}), t(\mathcal{P}), n(\mathcal{P})]$

FIGURE 1 A schematic view of the bidirectional pulse algorithm for the CSP.

3 | ACCELERATION STRATEGIES FOR THE BIDIRECTIONAL PULSE

In this section, we first review the core pruning strategies of the bidirectional pulse algorithm, namely infeasibility, bounds, and dominance pruning. Then, we present a detailed description of the proposed primal bound-update strategies. Finally, we discuss key aspects regarding the pulse queue and its exploration order.

3.1 | Pruning strategies

This section outlines the core strategies implemented for the CSP adapted from Lozano and Medaglia [16]. Sections 3.1.1, 3.1.2, and 3.1.3, provide details of lines 6, 7, and 8 in Algorithm 2, respectively.

3.1.1 | Pruning by infeasibility

The idea behind this strategy is to prune pulses as soon as it is known that it will not be possible to reach the destination node v_ω meeting the resource constraint. Thus, we can prune a partial path $\mathcal{P}(\alpha, i)$ arriving to node v_i in the search direction k when the cumulative resource consumption $t(\mathcal{P}(\alpha, i))$ already (meets or) exceeds the maximum resource consumption T. We can make this pruning strategy stronger by using the information of the resource consumption $t(\mathcal{P}^t(i, \omega))$ of the minimum resource consumption path $\mathcal{P}^t(i, \omega)$ from v_i to the destination node v_ω. More specifically, let $\bar{t}(v_i) = T - t(\mathcal{P}^t(i, \omega))$ be the resource consumption limit for a partial path arriving to any given node v_i. Thus, we prune partial path $\mathcal{P}(\alpha, i)$, if $t(\mathcal{P}(\alpha, i)) > \bar{t}(v_i)$ as this partial path cannot reach the destination node without violating the resource constraint.

3.1.2 | Pruning by bounds

In a similar fashion, we prune a pulse as soon as it is proven to have a cost equal or greater than the best objective $c(\mathcal{P}^*)$ found so far, i.e., we prune a partial path $\mathcal{P}(\alpha, i)$ if $c(\mathcal{P}(\alpha, i)) \geq c(\mathcal{P}^*)$. Furthermore, we strengthen this strategy by pruning pulses using the cumulative cost of the minimum cost path $c(\mathcal{P}^c(i, \omega))$ from v_i to the destination node v_ω. Given that $c(\mathcal{P}^c(i, \omega))$ is the best achievable cost from node v_i, we can safely prune a pulse if $c(\mathcal{P}(\alpha, i)) + c(\mathcal{P}^c(i, \omega) \geq c(\mathcal{P}^*)$.

3.1.3 | Pruning by dominance

During the recursive search, a node v_i could be reached more than one time by different pulses. With this in mind, we define dominance relations between two partial paths $\mathcal{P}_1(i)$ and $\mathcal{P}_2(i)$ arriving to node v_i on the same search direction κ. Partial path $\mathcal{P}_1$ *strongly dominates* $\mathcal{P}_2$ if:

$$c(\mathcal{P}_1(i)) < c(\mathcal{P}_2(i)) \quad \text{and} \quad t(\mathcal{P}_1(i)) < t(\mathcal{P}_2(i)).$$

Additionally, partial path $\mathcal{P}_1$ *weakly dominates* $\mathcal{P}_2$ if:

$$\begin{aligned} & c(\mathcal{P}_1(i)) = c(\mathcal{P}_2(i)) \quad \text{and} \quad t(\mathcal{P}_1(i)) < t(\mathcal{P}_2(i)) \\ \text{or} \quad & c(\mathcal{P}_1(i)) < c(\mathcal{P}_2(i)) \quad \text{and} \quad t(\mathcal{P}_1(i)) = t(\mathcal{P}_2(i)). \end{aligned}$$

We use the non-dominated list of labels $\mathcal{L}^k(v_i)$ to check for strong and weak dominance to prune incoming pulses to node v_i from both search directions independently. Since the number of labels on $\mathcal{L}^k(v_i)$ is limited by R is necessary to define a strategy to decide which labels are kept and which labels are discarded. We use a mixture between a random and an elitist rule. More specifically, at each node we keep a first non-dominated label that is overwritten every time that a partial path exhibits a lower cost than the one stored and a second non-dominated label that is overwritten every time that a partial path exhibits a lower resource consumption than the one stored. If $R > 2$, we randomly replace a label within the third and the R-th positions. For further discussion on the storage rules the interested reader is referred to Lozano and Medaglia [16].

3.2 | Primal bound-update strategies

In the context of the pulse algorithm, the strategy of pruning by bounds strongly depends on the quality of the primal bound. For this reason, we devise primal bound-update strategies with the purpose of strengthening these primal bounds as fast as possible. These strategies rely on exploring the network in both directions at the same time to increase the chances of finding high-quality solutions earlier in the execution of the algorithm, which in turns result in more effective pruning as a result of having tighter primal bounds earlier in the exploration. On the downside, they require additional computational resources for the initialization procedures and the execution of the strategies in both directions. Sections 3.2.1 and 3.2.2 provide details of lines 9 and 10 in Algorithm 2, respectively.

3.2.1 | Path completion

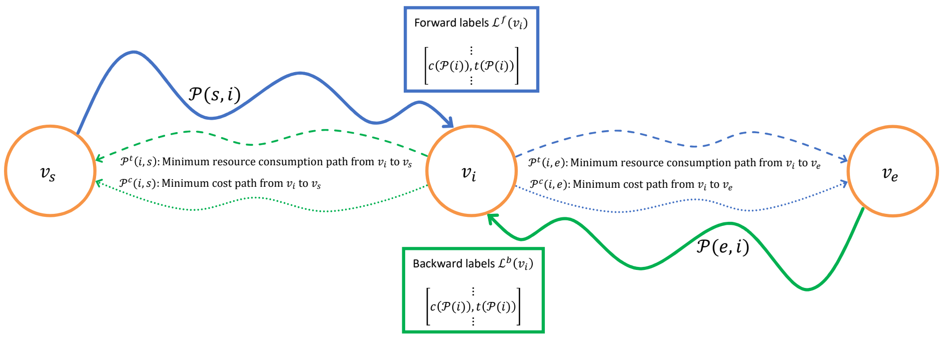
The main purpose of this strategy is to update the primal bound early in the exploration of the network. An additional benefit of this strategy is that it allows us in certain cases to prune partial paths as well. Using the minimum cost path or the minimum resource consumption path to reach the destination node v_ω, we test whether joining such a path with the partial path being explored improves the primal bound. Formally, let us consider a partial path $\mathcal{P}(\alpha, i)$ arriving to node v_i in the search direction k. The path completion strategy greedily adds the minimum cost path $\mathcal{P}^c(i, \omega)$ to the partial path $\mathcal{P}(\alpha, i)$, i.e., $\mathcal{P}(\alpha, \omega) = \mathcal{P}(\alpha, i) \cup \mathcal{P}^c(i, \omega)$. If the completed path $\mathcal{P}(\alpha, \omega)$ is feasible and $c(\mathcal{P}(\alpha, \omega)) < c(\mathcal{P}^*)$, we update the incumbent solution accordingly. Furthermore, we can prune the incoming pulse $\mathcal{P}(\alpha, i)$, because (by construction) we know that $\mathcal{P}(\alpha, \omega)$ is already the minimum cost path beginning with the partial path $\mathcal{P}(\alpha, i)$.

In case the minimum cost path cannot be successfully added to the current partial path, there is still a chance to update the incumbent solution. Given a partial path $\mathcal{P}(\alpha, i)$, the path can be completed by adding the minimum resource consumption path $\mathcal{P}^t(i, \omega)$ to the partial path $\mathcal{P}(\alpha, i)$, i.e., $\mathcal{P}(\alpha, \omega) = \mathcal{P}(\alpha, i) \cup \mathcal{P}^t(i, \omega)$. If the completed path $\mathcal{P}(\alpha, \omega)$ is feasible and $c(\mathcal{P}(\alpha, \omega)) < c(\mathcal{P}^*)$, the incumbent solution can be updated. Note, however, that the associated pulse $\mathcal{P}(\alpha, i)$ cannot be pruned because there is still a chance to find a completion path from v_i to v_ω with a cost lower than $c(\mathcal{P}(\alpha, \omega))$.

3.2.2 | Path joins

Formally, given a partial path $\mathcal{P}(\alpha, i)$ arriving to node $v_i \in \mathcal{N}$ in the search direction k, the path joins strategy adds all partial paths stored on $\mathcal{L}^{k'}(v_i)$, which are the partial paths that arrived at node v_i from the opposite direction k'. Consider a partial path $\mathcal{P}(\omega, i)$ coming from the destination node ω to node i, and let $\texttt{reverse}(\mathcal{P}(\omega, i))$ be its reversed counterpart (i.e., the same sequence of nodes, but in reverse order). Extending the partial path $\mathcal{P}(\alpha, i)$ with $\texttt{reverse}(\mathcal{P}(\omega, i))$ creates a complete path $\mathcal{P}(\alpha, \omega)$, i.e., $\mathcal{P}(\alpha, \omega) = \mathcal{P}(\alpha, i) \cup \texttt{reverse}(\mathcal{P}(\omega, i))$. If the complete path is feasible and $c(\mathcal{P}(\alpha, \omega)) < c(\mathcal{P}^*)$, the incumbent solution is updated. Note that this strategy entirely depends on the bidirectional search to be performed in parallel as partial paths can only be joined if the search from the opposite direction has already stored labels at node v_i. The order in which nodes are visited and the quality of the (limited) labels stored at each node can be controlled to some extent by the order in which outgoing arcs are explored at each node. We adopted an ordering rule in which outgoing arcs of each node are sorted in ascending order according to a metric that sums the cost of an arc plus the (minimum) cost-to-go from the corresponding head node.

Figure 2 shows a summary of the proposed primal bound-update strategies. Using the path joins and the path completion strategies enable us to construct feasible paths without reaching the corresponding destination node, increasing the chances of finding high-quality feasible solutions early in the exploration. We delay until §6, the introduction of a simple, yet effective heuristic that takes advantage of this fact. Furthermore, our computational experiments in §4, show the success of our primal bound-update strategies to identify near-optimal solutions very early in the recursive exploration on a set of benchmark instances. In Appendix B we include a numerical example that illustrates both the pruning and the primal bound-update strategies.


Forward labels $\mathcal{L}^f(v_i)$
$\begin{bmatrix} \vdots \\ c(\mathcal{P}(i)), t(\mathcal{P}(i)) \\ \vdots \end{bmatrix}$
$\mathcal{P}(s,i)$
v_s
v_i
v_e
$\mathcal{P}^t(i,s)$: Minimum resource consumption path from v_i to v_s
$\mathcal{P}^c(i,s)$: Minimum cost path from v_i to v_s
$\mathcal{P}^t(i,e)$: Minimum resource consumption path from v_i to v_e
$\mathcal{P}^c(i,e)$: Minimum cost path from v_i to v_e
$\mathcal{P}(e,i)$
Backward labels $\mathcal{L}^b(v_i)$
$\begin{bmatrix} \vdots \\ c(\mathcal{P}(i)), t(\mathcal{P}(i)) \\ \vdots \end{bmatrix}$

$c(\mathcal{P}^*)$ is updated by path $\mathcal{P}(\alpha, i)$ $\begin{cases} \text{Minimum cost path completion if } c(\mathcal{P}(\alpha,i)) + c(\mathcal{P}^c(i,\omega)) < c(\mathcal{P}^*) \wedge \mathcal{P}(\alpha,\omega) \text{ is feasible} \\ \text{Minimum resource consumption path completion if } c(\mathcal{P}(\alpha,i)) + c(\mathcal{P}^t(i,\omega)) < c(\mathcal{P}^*) \wedge \mathcal{P}(\alpha,\omega) \text{ is feasible} \\ \text{Path joins if } \exists \text{ label } l \in \mathcal{L}^{\kappa'}(v_i) \text{ such that } c\big(\mathcal{P}(\alpha,i)\big) + c\big(\mathcal{P}(\omega,i)\big) < c(\mathcal{P}^*) \wedge \mathcal{P}(\alpha,\omega) \text{ is feasible} \end{cases}$

FIGURE 2 Primal bound-update strategies

3.3 | Pulse queueing

The choice between a depth-first search (DFS) or a breadth-first search (BFS) strategy is non-trivial since the performance of each strategy is instance dependent. We adapt the queueing mechanism proposed by Bolívar et al. [21] for the bidirectional search by combining both search behaviors with a pulse queue Q^k for each search direction k. The key idea of these queues is to halt pulses by limiting the depth of the propagation in order to avoid exploring too deep into unpromising regions of the solution space. We denote by *steps* the number of times a pulse propagates, which corresponds to the number of additional nodes visited by the corresponding partial path since its exploration began (or was resumed after being halted inside a queue). We impose a maximum depth limit δ that restricts the number of steps that a pulse can advance before being halted and stored in the queue. Once there are no active pulses left, the search resumes the exploration of the queued pulses.

The queue discipline for each Q^k is critical for the algorithm's performance because it defines the graph exploration order. Given a partial path $\mathcal{P}(\alpha, i)$ arriving to node v_i, in the search direction k, we consider a *best promise* queueing discipline, where we define the best promise of a partial path as $\psi(\mathcal{P}(\alpha,i)) = c(\mathcal{P}(\alpha,i)) + c(\mathcal{P}^c(i,\omega)))$. The idea behind this queue discipline is to explore first those partial paths which promise the best possible objective function, i.e., those paths with a better chance of improving the incumbent, which correspond to paths exhibiting minimum $\psi(\mathcal{P}(\alpha,i))$. In particular, the operation $\mathrm{pop}(Q^k)$ in line 5 of Algorithm 1 removes and returns a partial path $\mathcal{P}$ from Q^k, such that $\psi(\mathcal{P})$ is the minimum value across all paths in Q^k.

4 | COMPUTATIONAL EXPERIMENTS

We study the performance of our proposed algorithm in two sets of experiments. Section 4.1 presents the results for the first set of experiments conducted over a testbed of large-scale road networks from the literature. Section 4.2 presents the results for the second set of experiments, in which our algorithm is embedded in a CG scheme to solve a multi-activity shift scheduling problem.

4.1 | Experiments over large-scale road networks

We compare the performance of our proposed bidirectional pulse algorithm (labeled by "BP") against the original pulse algorithm by Lozano and Medaglia [16] (labeled by "PA") and the state-of-the-art algorithm by Thomas et al. [17] (labeled by "RC-BDA"). The proposed algorithm "BP" and the original pulse algorithm "PA" were implemented in Java, compiled using Eclipse SDK version 4.8.0, and the experiments executed on a computer with an Intel Core i7-4610M @3.00 GHz with 8GB of RAM allocated to the memory heap size of the Java Virtual Machine on Windows 10. The initialization step of the PA and BP is conducted in parallel for each search direction.

To compare the algorithms, we use the large-sized US-road-network instances of the 9^{th} DIMACS challenge [28]. Each instance is a combination of a road network, a destination, and a tightness factor p. Low (high) values of p indicate that the resource constraint is tight (loose). We considered nine road networks, with five destinations each, and eight values of p ranging from 0.1 to 0.8 (step of 0.1), for a total of 360 instances. Information regarding the origin, the destination, the time limit T and the road network for each instance are available on `https://github.com/copa-uniandes/BidirectionalPulse`. Table 1 describes the road networks and its size in terms of the number of nodes and arcs.

TABLE 1 Large-sized US-road networks information.

	Road network	Nodes	Arcs
BAY	San Francisco Bay Area	321,270	800,172
NY	New York City	264,346	733,846
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
NE	Northeast USA	1,524,453	3,897,636
CAL	California and Nevada	1,890,815	4,657,742
LKS	Great Lakes	2,758,119	6,885,658
E	Eastern USA	3,598,623	8,778,114
W	Western USA	6,262,104	15,248,146

We use as a benchmark the computational results for the RC-BDA as reported in Thomas et al. [17]. They coded their algorithm in a PC running 64-bit Ubuntu 14.04 with a quad core 2.5 GHz Intel i7-4710HQ processor with 8GB of RAM. For the sake of fairness, we scaled our times by 1.12 according to the LINPACK benchmark [29] which takes into account differences in processing power. However, we acknowledge the difficulty to measure differences in operating systems and programming languages and as such, our computational results comparing BP and RC-BDA should be interpreted as a general indication of how the algorithms perform and not as a head-to-head performance comparison. On the contrary, the comparison between BP and PA is indeed a head-to-head race executed under identical conditions in the same machine.

Table 2 compares the performance of the bidirectional pulse algorithm against the benchmark algorithms. Each row corresponds to a road network from the US presented in Table 1. Columns 2, 4, and 6, show the average runtime in seconds for each algorithm over the solved instances (out of the 40 combinations of tightness and destination) per road network. Columns 3, 5, and 7, show the number of instances solved in less than 14,400 seconds (4 hours) by each algorithm (out of 40). After fine tuning the bidirectional pulse algorithm, the depth limit was set to ($\delta = 2$) and the memory size for each node was set to three labels ($R = 3$).

TABLE 2 Computational results on large-sized US-road networks

Road network	PA		RC-BDA		BP	
	Average time (s)	Solved	Average time (s)	Solved	Average time(s)	Solved
BAY	25.57	40/40	1.27	40/40	0.30	40/40
NY	100.82	40/40	0.59	40/40	0.23	40/40
COL	3.99	36/40	4.60	40/40	1.67	40/40
FLA	120.72	24/40	62.32	40/40	2.64	40/40
NE	16.21	35/40	4.06	40/40	1.24	40/40
CAL	21.90	28/40	643.87	40/40	109.93	35/40
LKS	5.08	24/40	816.74	37/40	319.26	36/40
E	9.30	29/40	65.08	40/40	13.63	40/40
W	1.01	16/40	1,063.44	40/40	139.59	38/40
		272/360		357/360		349/360

*Average time on solved instances

Table 2 shows that the proposed BP solves 349 out of the 360 instances within the time limit. Furthermore, the BP solves 87 more instances than the original PA, thus showing a remarkable improvement in terms of scalability. In contrast, the RC-BDA solves slightly more instances (eight) than the BP. In terms of average times, these times vary widely depending on the road network. However, we can see qualitatively that the proposed BP behaves much better than the PA, and it is competitive against RC-BDA. Note that the original PA is indeed fast on those instances where it can find a solution, yet it fails to find one in 88 instances, within the time limit. At the other end of the spectrum, the average time over the solved instances of RC-BDA is larger than BP, yet it finds eight more solutions on harder instances of the CAL, LKS, and W road networks, within the time limit.

Table 3 shows average speedups for the BP computed as the ratio between the execution time of the BP and the execution time of the corresponding benchmark algorithm. For each road network, columns 2 and 5 show the arithmetic mean of the speedups of the BP against each benchmark algorithm. In addition, columns 3 and 6 show the geometric mean of the speedups of the BP against both benchmark algorithms. Note that opposed to the arithmetic mean, the geometric mean avoids being overly optimistic with large ratios obtained on few instances, thus it provides a fairer comparison [30]. Columns 4, and 7, present the number of instances in which the BP was faster than the corresponding benchmark algorithm.

Remarkably, Table 3 shows that the proposed BP is consistently faster in most of the instances than the benchmark approaches and achieves considerable speedups over all road networks. Specifically, the BP is roughly 750 and 13 times faster in average than the pulse algorithm (PA) considering the arithmetic and the geometric mean, respectively. However, the PA is faster in more than half of all instances (212 out of 360 instances). We explain this by noting that the PA exhibits an "all-or-nothing" type of behavior as when it solves an instance, it usually solves it very fast; but it also fails to solve a considerable number of instances within the time limit adding a lot of variability to the solution times. In comparison with the RC-BDA, the BP is roughly 12 and 4 times faster in average considering the arithmetic and the geometric mean, respectively. Furthermore, the BP is faster in 335 out of the 360 instances.

TABLE 3 Average and geometric speedups vs benchmark algorithms on large-size US-road networks

Road network	PA			RC-BDA		
	Arithmetic mean of speedups	Geometric mean of speedups	BP Wins	Arithmetic mean of speedups	Geometric mean of speedups	BP Wins
BAY	27.88	1.03	9/40	4.42	2.89	36/40
NY	415.34	3.93	18/40	2.62	2.33	38/40
COL	126.58	1.37	7/40	3.61	2.79	40/40
FLA	3117.64	69.33	24/40	25.82	8.35	40/40
NE	1106.97	3.56	16/40	3.40	3.07	40/40
CAL	371.62	3.81	19/40	12.59	4.76	35/40
LKS	321.56	5.53	20/40	12.03	5.16	35/40
E	553.00	4.82	13/40	5.13	2.20	34/40
W	719.34	23.40	22/40	37.59	7.19	37/40
Overall avg.	751.10	12.97	148/360	11.91	4.30	335/360

*The speedup calculation assigns a computational time of 14,400 seconds for unsolved instances.

4.2 | Solving pricing problems stemming from a CG approach

We study a multi-activity shift scheduling problem (MASSP), which aims to find a set of shifts to satisfy a demand for staff requirements over a planning horizon [8, 31]. An employee may perform different work activities in the same shift, where $\mathcal{K}$ denotes the set of activities. We consider upper limits, u_i, that constraint the number of time periods that an employee can work on each activity as well as the number of times that an employee can change between activities during the shift.

We explore a CG approach to solve the linear relaxation of the problem. Let Ω be the set of all feasible shifts (columns) and let integer variable λ_j be the number of employees assigned to shift j. Let binary matrix $\mathbf{s}^j$ denote the schedule for shift $j \in \Omega$, where $s^j_{it} = 1$ indicates that an employee is assigned to work activity $i \in \mathcal{K}$ at time $t \in \{1, \dots, H\}$ of the planning horizon. Let d_{it} denote the staffing requirement for activity i at time t. The MASSP can be formulated as the following *master problem*:

$$\text{MP}: \quad \min \quad \sum_{j \in \Omega} \lambda_j \tag{1a}$$

$$\text{s.t.} \sum_{j \in \Omega} s^j_{it} \lambda_j \geq d_{it} \qquad \forall i = 1, \dots, |\mathcal{K}|,\ t = 1, \dots, H \tag{1b}$$

$$\lambda \in \mathbb{Z}^{|\Omega|}_{+}. \tag{1c}$$

The objective function (1a) minimizes the total number of employees hired. Constraints (1b) ensure that the staffing requirements are satisfied per activity and time period. Constraints (1c) require the λ-variables to be nonnegative integers. We refer to the linear relaxation of model (1) as the *relaxed master problem*. Feasible shifts in set Ω must satisfy the following constraints:

- at most one work activity can be performed during any time period;

- an employee can work at most u_i time periods on activity i during a shift;
- there is no idle time once the shift starts; and
- switching activities takes one period of time.

As the size of set Ω could be exponentially large, we consider a CG approach that generates shifts (columns) iteratively by solving a pricing problem (ofen called auxiliary problem), which searches for a minimal reduced cost shift (column). Let π be the dual variables associated with constraints (1b) in the relaxed master problem. Let x_{it} be a binary variable that takes the value of 1 if activity i is scheduled at time t; it takes the value of 0, otherwise. Let y_t be a binary variable that denotes the start of the shift, and it takes the value of 1 if the shift starts at time t; it takes the value of 0, otherwise. We let activity 1 denote that the employee is switching activities and thus u_1 denotes the maximum number of times that an employee can change between work activities during a shift. A standard integer programming formulation for the pricing problem is:

$$\text{PP}: \quad \min 1 - \sum_{i=1}^{|\mathcal{K}|}\sum_{t=1}^{H} x_{it}\pi_{it} \tag{2a}$$

$$\sum_{i=1}^{|\mathcal{K}|} x_{it} \leq 1 \qquad \forall t = 1,\dots,H \tag{2b}$$

$$\sum_{t=1}^{H} x_{it} \leq u_i \qquad \forall i = 1,\dots,|\mathcal{K}| \tag{2c}$$

$$\sum_{t=1}^{H} y_t = 1 \tag{2d}$$

$$x_{i1} \leq y_1 \qquad \forall i = 1,\dots,|\mathcal{K}| \tag{2e}$$

$$x_{it} \leq y_t + x_{it-1} + x_{1t-1} \qquad \forall i = 2,\dots,|\mathcal{K}|,\ t = 2,\dots,H \tag{2f}$$

$$x_{1t} \leq \sum_{i=2}^{|\mathcal{K}|} x_{it-1} \qquad \forall t = 2,\dots,H \tag{2g}$$

$$\mathbf{x} \in \{0,1\}^{|\mathcal{K}|\times H};\ \mathbf{y} \in \{0,1\}^{H}. \tag{2h}$$

The objective function (2a) minimizes the reduced cost of the shift. Constraints (2b) ensure that at most one work activity is performed at each time period. Constraints (2c) enforce the upper time limits for each time activity. Constraint (2d) requires that the shift starts exactly once. Constraints (2e)–(2g) enforce shift continuity. Note that constraints (2g) ensure that an activity change only occurs when the employee is busy in the previous time period. Our CG approach iteratively solves the relaxed master problem and the pricing problem until no negative reduced shift (column) exists ensuring an optimal solution to the linear relaxation of the MASSP.

Problem (2) can be transformed into a CSP with multiple resources over a directed acyclic network, where each resource corresponds to a work activity as shown in Appendix A. To solve this variant of the CSP, we modified the bidirectional pulse algorithm as follows: (1) in the initialization step we found lower bounds on the minimum cost and all the minimum resources needed to reach the end node; (2) the infeasibility pruning strategy was extended to prune using multiple resource constraints; (3) the path completion strategy completes paths using minimum-resource completions for every resource; and (4) for the path joins strategy we need to check that all resource constraints are met. All other acceleration strategies remain the same.

We compare the performance of our proposed algorithm against a traditional single-directional labeling approach denoted by "LA" [32] inside our CG scheme over a set of 15 synthetic instances. We consider a number of activities $|\mathcal{K}|$ ranging from three to five (including the change between activities modeled as activity 1) and generate five random instances for each value of $|\mathcal{K}|$. We generate upper time limits u_i for each activity using a discrete uniform distribution between 2 and 5. In addition, we generate staffing demands d_{it} following a uniform distribution between 10 and 20. Finally, the number of time periods for all instances was set to 100.

Table 4 compares the performance of the bidirectional pulse algorithm against the labeling algorithm while solving the auxiliary problem of the CG procedure. Each row corresponds to a number of activities, i.e., results are averaged over the five instances generated for each value of $|\mathcal{K}|$. We solve the relaxed master problem to optimality and report the maximum, average, and minimum time (in seconds) it takes to solve one pricing problem to optimality during the execution of the CG procedure. Columns 2 and 6 show the maximum runtime for solving a pricing problem for each algorithm. Columns 3 and 7 show the average runtime while columns 4 and 8 show the average minimum runtime. Finally, columns 5 and 9 show the number of relaxed master problem instances solved to optimality by the CG within a time limit of $3,600$ seconds (1 hour) using each algorithm.

TABLE 4 Comparison of LA vs BP as the auxiliary problem under CG for MASSP instances

$\|\mathcal{K}\|$	LA				BP			
	Max time (s)	Average (s)	Min time (s)	Solved	Max time (s)	Average (s)	Min time (s)	Solved
3	0.38	0.11	0.01	5/5	0.18	0.03	<0.01	5/5
4	33.47	2.42	0.16	4/5	1.22	0.34	0.01	5/5
5	73.27	27.56	5.63	4/5	13.04	3.04	<0.01	5/5
Overall avg.	35.71	10.03	1.93	13/15	4.81	1.13	0.01	15/15

Table 4 shows that using our algorithm embedded in a CG scheme allows us to solve all instances to optimality within the time limit. In contrast, the CG scheme that uses the LA is unable to solve two instances. In terms of the average time per pricing problem, the BP outperforms LA both in terms of the maximum and average solution times. Our proposed BP is roughly 4, 7, and 9 times faster than the LA for instances with 3, 4, and 5 activities, respectively. This behaviour is explained by the fact that the LA is heavily dependent on dominance relationships to eliminate suboptimal paths. As the number of activities increases, dominance relationships become weaker and it is more computationally expensive to check if partial paths are dominated. In contrast, the BP uses a combination of several pruning strategies along with path completion that enable the algorithm to accelerate the search.

5 | SENSITIVITY ANALYSIS OF THE BIDIRECTIONAL PULSE

We conducted three experiments to further assess the performance of the bidirectional pulse. First, we propose an experiment to evaluate the impact of performing a bidirectional search against a single-directional search on the CSP. In contrast to the assessment made in Section 4.1, where the original PA does not implement the path completion and pulse queue strategies, the single-directional search variants shown in this section embed these strategies. Second, we conducted an experiment on the relative effectiveness of the pruning and the primal bound-update strategies. Finally, we conducted a sensitivity analysis on the value of the maximum depth limit δ for the pulse queues. For these experiments, we chose the road networks of San Francisco Bay Area, New York City, Colorado, Florida, and Eastern USA, as these networks are a representative sample of the original testbed with a mixture of smaller and larger networks.

5.1 | Bidirectional versus single-directional search

To analyze the impact of exploring the search space in both directions we ran all 40 instances (combinations of tightness and destination) on each of the selected five networks, performing a single-directional search, sending pulses either in the forward direction or the backward direction (including all strategies outlined in §3). Table 5 compares the performance of each algorithm. Columns 2, 3, and 4 show the average computational times achieved with the BP, a forward-direction-only pulse algorithm (FDP) and a backward-direction-only pulse algorithm (BDP), respectively. In addition, columns 5 and 6, show the speedups achieved by the BP against the single-directional pulses. Finally, column 7 shows the speedup of the FDP against the BDP. To calculate these speedups, we first computed the speedup achieved on each of the 40 instances (i.e., pairwise or head-to-head) for each road network. Then, we calculated the geometric mean.

TABLE 5 Bidirectional pulse vs single-directional pulse on large-size road networks

Road network	BP (s)	FDP (s)	BDP (s)	BP vs FDP speedup	BP vs BDP speedup	FDP vs BDP speedup
BAY	0.30	0.26	0.79	0.77	2.61	3.11
NY	0.23	0.85	0.20	3.52	0.85	0.24
COL	1.67	1.61	2.31	0.93	1.42	1.53
FLA	2.64	1.98	14.35	0.78	5.21	7.32
E	13.63	239.26	12.94	10.30	0.94	0.05
Overall avg.	3.69	48.79	6.12	3.26	2.21	2.45

We also tracked which search direction finished first in the BP. Figure 3 shows the fraction of instances in which the search in either the forward or the backward direction terminates first (therefore stopping the execution of the algorithm), considering the 40 instances of each of the five road networks.

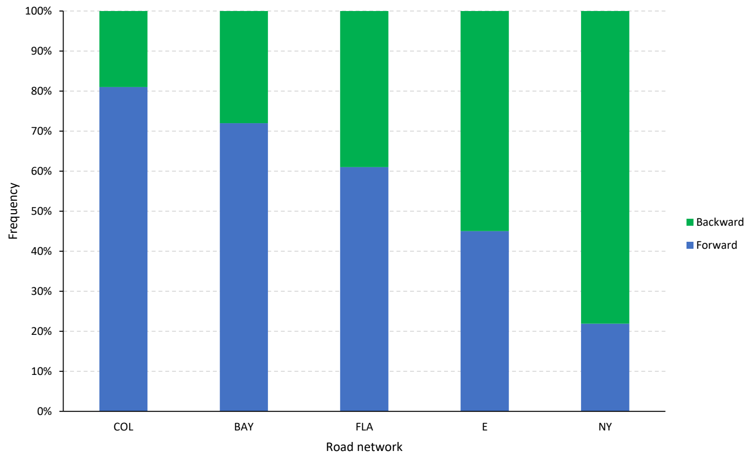


FIGURE 3 Fraction of instances in which the forward and the backward search finished first

Both Table 5 and Figure 3 show that the performance of a particular search direction varies from one instance to another. As a practical matter, we never know beforehand which search direction would be better. However, these results are insightful as they highlight the robustness of the bi-directional approach relative to the single-directional approaches. As BP explores both directions concurrently, the computational time of the BP is roughly the time of the best single-directional approach (FDP or BDP) plus a small overhead (see Table 5). This is expected because the BP terminates as soon as one of the searches in any direction is completed.

5.2 | Relative effectiveness of the bidirectional pulse acceleration strategies

We designed two experiments to assess the relative contribution of the acceleration and pruning strategies to the overall performance of the bidirectional pulse algorithm. We refer the reader to Bolívar et al. [21] for an analysis of the relative contribution of the acceleration strategies (path completion and pulse queue) over the original core strategies of the PA [16]. For both experiments, we ran all 40 instances (combinations of tightness and destination) on each of the selected five networks.

In the first experiment, we analyzed which strategies update the primal bound more frequently. Figure 4 shows that the minimum resource consumption path completion in both the forward (MTF) and backward (MTB) direction are the ones that update the incumbent more frequently. Moreover, the algorithm relies on the minimum resource consumption path completion strategy to update the primal bound about 70% of the time. Note that the path joins (PJ) strategy seems to have greater impact when the resource constraint is tighter (smaller *p* values). On the contrary, regardless of the search direction, the minimum-cost path completion relative effectiveness increases when the resource constraint is loose (larger *p* values).

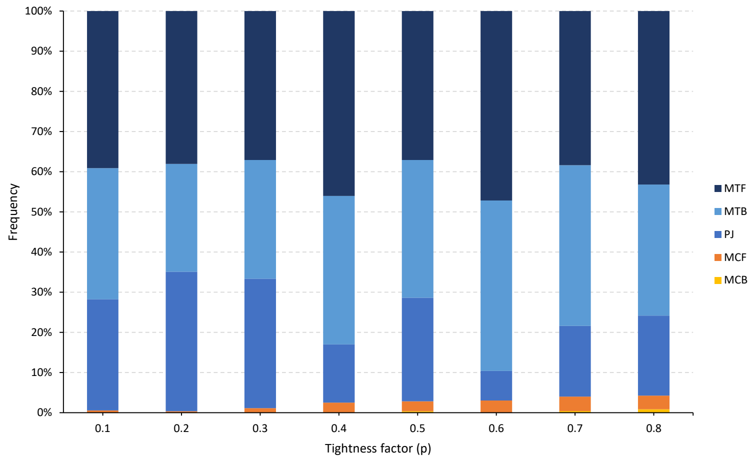


FIGURE 4 Relative effectiveness of the primal bound-update strategies

In the second experiment, we analyzed the relative effectiveness of the pruning strategies based on how frequently they pruned partial paths. Figure 5 shows the relative effectiveness of the pruning strategies as the tightness factor varies. In summary, the most effective strategy is dominance pruning. More specifically, the algorithm relies on this strategy to prune almost 80% of the partial paths, followed by bounds pruning with 15%, and infeasibility pruning with just 5%. Nonetheless, since pruning a path at an early stage might be more effective than a very frequent pruning strategy used at a later stage, this analysis only measures the relative effectiveness of the strategies. Note that as the resource constraint becomes loose, the relative effectiveness of the bounds pruning strategy increases.

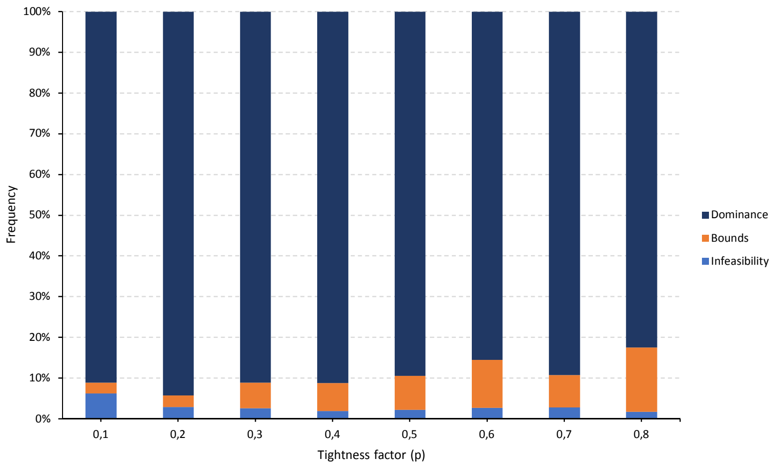


FIGURE 5 Relative effectiveness of the pruning strategies

5.3 | The effect of halting pulses: queues and their depth limit

Finally, we studied the impact of the maximum depth limit δ (of the pulse queues) on the BP performance. With this purpose we ran all 40 instances (combinations of tightness and destination) on each of the selected five networks. Then, we used the performance degradation proposed by Bolívar et al. [21], defined as the ratio of the average time (for a given δ) to the minimum average time (across all values of δ). A performance degradation of 1 means that the given depth limit δ achieved the minimum average time (as good as it gets). Figure 6 shows the performance degradation as a function of the depth limit over each road network.

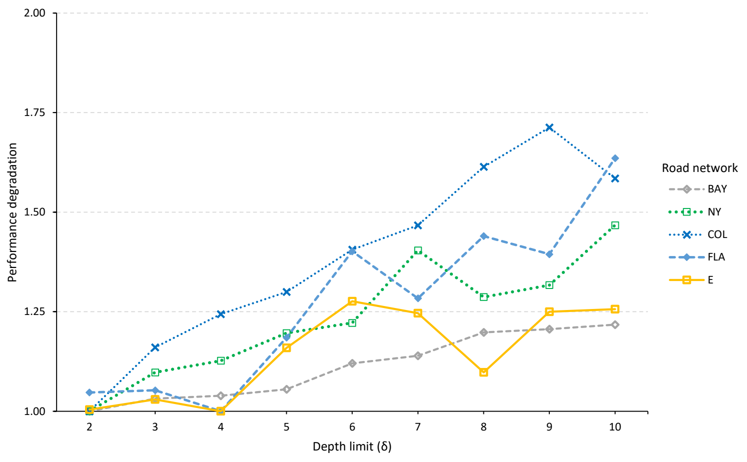


FIGURE 6 Sensitivity analysis on the depth limit on a sample of large-sized US-road networks

For almost all the road networks considered, the BP achieved its best performance with a depth limit of two. Note that an increase on the depth limit favors depth over breadth, and in these road networks, it degrades the performance of the bidirectional pulse. The short depth allows the bidirectional pulse to re-route the search more frequently, without going too deep into the network when it is too late to backtrack.

6 | A PULSE-BASED HEURISTIC FOR THE CSP

The bidirectional pulse is an exact algorithm for the CSP. However, in some applications finding an optimal solution is not mandatory. For example, when the CSP is used as the auxiliary problem on a column generation scheme (as shown in §4.2), it is only necessary to find paths with negative reduced cost during the early iterations of the scheme. For this reason, we analyze the ability of the bidirectional pulse to find high quality solutions during the initial stages of the exploration using the primal bound-update strategies on the California and the Great Lakes road networks. We chose these networks because all algorithms struggled to solve all instances to optimality (within the time limit of four hours), and for those they solved, they took longer (on average) than on the other road networks. In particular, RC-BDA and BP, failed to solve 3 and 9 instances out of 80 on these hard road networks within the time limit. Figures 7 and 8 show the computational time required to find a solution within a 1% gap and the computational time needed to find an optimal solution for the California and the Great Lakes road networks instances, respectively. For this experiment, we used the same machine specifications of §4 and imposed a time limit of 14, 400 seconds. In order to compute optimality gaps, we found optimal solutions for the unsolved instances from §4 using a machine with Intel Xeon E5-2630 v3 @2.40 GHz with 14 GB allocated on the memory heap size of the Java Virtual Machine and no time limit.

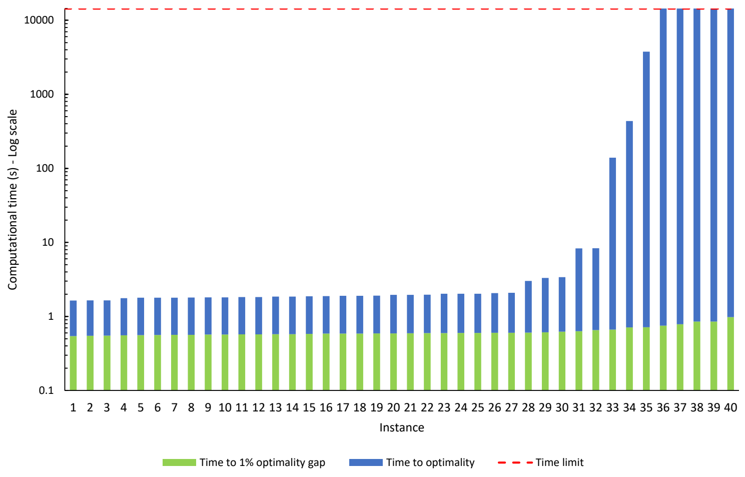


FIGURE 7 Performance of the bidirectional pulse on the California road network instances (sorted by solution time)

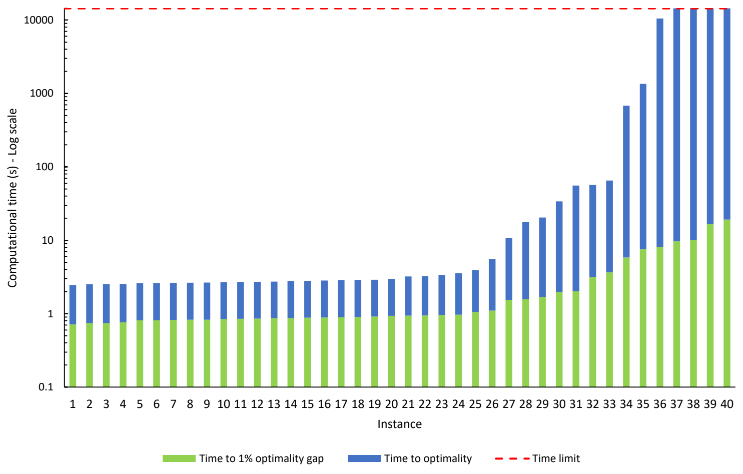


FIGURE 8 Performance of the bidirectional pulse on the Great Lakes road network instances (sorted by solution time)

Note that in all the California road network instances, the bidirectional pulse found a solution within a 1% gap in less than one second. However, the BP was not able to prove optimality within the time limit for five instances (out of 40). Similarly, for the Great Lakes instances, the BP found solutions within a 1% gap in less than 20 seconds. However, the algorithm failed to close the gap for four of those 40 instances within the time limit. We note that, as it is often the case in discrete problems, the algorithm finds high quality solutions in initial stages of the search and spends most of the computational time closing the optimality gap.

To complement our analysis, we selected a hard instance of the Western USA road network (destination 5 and tightness factor of 0.2) based on the computational times that both the BP and the RC-BDA needed to solved it (RC-BDA took more than two hours). Then, we ran the BP storing information about which strategy updated the primal bound and the values for the sequence of bounds. Figure 9 shows the primal bounds, the time at which each solution was found, and the strategy that found the solution. As a reference, the optimal cost is marked by a green horizontal line. The initial solution is the cost of the minimum resource consumption path found on the initialization step. In addition, we denote a minimum resource consumption path as MTF or MTB, depending on the search direction, and the path joins strategy as PJ. Note that on this instance the minimum resource consumption path completion on the backward direction (MTB) found solutions faster. However, the minimum resource consumption path completion on the forward direction (MTF) was able to construct solutions of higher quality. Moreover, on this instance the BP was not able to complete paths using the minimum-cost path completion. In addition, is relevant to mention that an optimal solution was found by the path joins (PJ) strategy, 110 seconds after the algorithm began. Finally, note that 900 seconds elapsed from the moment in which an optimal solution was found until BP terminates. Although this analysis is instance-dependent, we observed a similar behavior on several instances of the testbed, where the BP found an optimal solution at an early stage but needed considerable additional time to prove optimality.

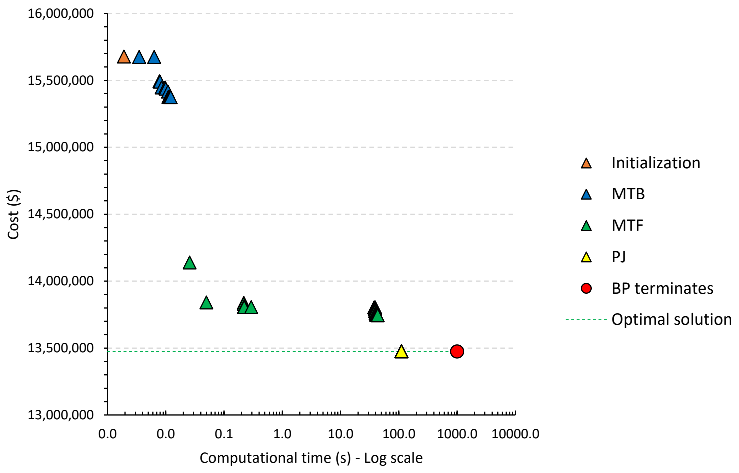


FIGURE 9 Primal-bound evolution for BP on the Western USA road network on the destination 5 with 0.2 tightness

Based on the previous analysis, in some applications it might be worth to design a pulse-based heuristic for the CSP just by adding a naïve stopping criterion to the BP. We conducted an experiment where we impose a computational time limit γ. If the computational time reaches this value γ, the algorithm stops, and we accept the current incumbent solution. To evaluate the quality of the solutions obtained with this pulse-based heuristic, we used the complete testbed of 360 instances introduced in §4, varying the time limit γ. Figure 10 shows the number of instances where the heuristic found a solution within a 1% gap for a given computational time budget γ. Note that 90%, 95%, and 99% of all instances are within 1% of the optimal solution in less than 3, 5, and 30 seconds, respectively.

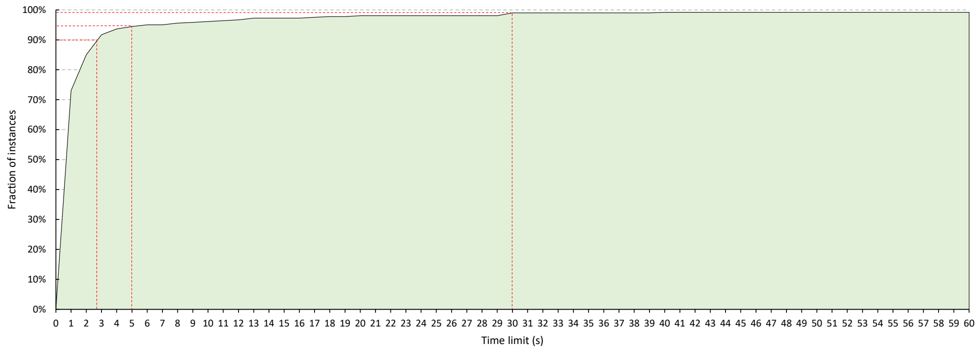


FIGURE 10 Quality of the pulse-based heuristic as the fraction of instances within 1% optimality as a function of the computational budget (time limit) for the large-size US-road networks

7 | CONCLUDING REMARKS

In this work we presented an exact algorithm for the CSP that solves large-sized road networks with up to $6,000,000$ nodes and $15,000,000$ arcs. The algorithm is based on a bidirectional adjustable depth-first (or breadth-first) search leveraged by parallelism and a set of acceleration strategies. Specifically, we incorporated core pruning strategies that allow to prune partial paths at early stages of the search. In addition, we integrated the path completion and the path joins strategies to update the incumbent solution in intermediate parts of the network. It is noteworthy that the only parameters to be defined by the user are the maximum number of labels stored at each node (R) and the depth limit (δ) that adjusts the search behavior (balancing between a breadth or depth emphasis).

From a computational perspective, we conducted several experiments over 360 instances from nine large-scale road networks in the US to test the performance of the proposed algorithm against two state-of-the-art algorithms, namely, the (original) pulse algorithm –PA– and the bidirectional A* algorithm –RC-BDA–. Against PA, the proposed bidirectional pulse algorithm reached average speedups of up to 70 times and was on average 12 times faster according to the conservative geometric mean. Moreover, the bidirectional pulse algorithm was able to solve 87 instances more than the PA within the 4-hour time limit. On the other hand, compared with the RC-BDA, the bidirectional pulse was on average four times faster, even though the RC-BDA solved 8 instances more. Additionally, we embedded our algorithm in a column generation scheme to solve a multi-activity shift scheduling problem, where the auxiliary problem can be represented as a CSP problem with multiple resources. We compared the performance of the BP against a traditional single-directional labeling algorithm for solving the auxiliary problem, where the BP showed a remarkable performance both in terms of the maximum and average solution times. In summary, the bidirectional pulse algorithm presents a fast, reliable, and stable exact algorithm for the CSP with one or multiple resources.

In addition, we conducted a comprehensive sensitivity analysis to better understand the algorithm components and their relative contribution to the bidirectional pulse. First, we showed the impact of performing a bidirectional search comparing the algorithm against a single-directional version of the algorithm. Then, we designed an experiment to test the primal bound-update strategies in which we observed that the path joins strategy has a major relevance when the resource constraint is tighter. Also, we noted that the minimum resource consumption path completion is the most used strategy to create feasible solutions at early stages of the search. In addition, we analyzed the impact of halting pulses on the bidirectional search, varying the pulse depth limit, noting that a smaller value leads to better results (favoring breadth over depth, in this case).

Finally, although the bidirectional pulse is an exact method for the CSP, we derived a new heuristic for the CSP that finds high quality solutions for time-constrained applications –like in the pricing problem within a column generation scheme–. We showed that the heuristic can find solutions within 1% of optimality in less than 30 seconds for 99% of the 360 instances of the testbed.

We are currently working on an extension that incorporates uncertainty on the arc weights and deals with chance constraints. We are also evaluating the benefits versus the computational overhead of using multiple threads on each search direction.

ACKNOWLEDGEMENTS

The authors are grateful for the detailed remarks of two anonymous referees, which greatly helped improve the manuscript. Dr. Lozano gratefully acknowledges the support of the Office of Naval Research under Grant N00014-19-1-2329.

REFERENCES

[1] Garey MR, Johnson DS. Computers and intractability: A guide to the theory of NP-Completeness. New York, NY, USA: W. H. Freeman & Co.; 1979.

[2] Zabarankin M, Uryasev S, Pardalos P. Optimal risk path algorithms. Cooperative Control and Optimization 2005;p. 273–298.

[3] Cabral EA, Erkut E, Laporte G, Patterson RA. The network design problem with relays. European Journal of Operational Research 2007;180(2):834–844.

[4] Desaulniers G, Desrosiers J, Solomon MM. Column generation. Boston, MA, USA: Springer; 2005.

[5] Derigs U, Friederichs S, Schäfer S. A new approach for air cargo network planning. Transportation Science 2009;43(3):370–380.

[6] Graves GW, McBride RD, Gershkoff I, Anderson D, Mahidhara D. Flight crew scheduling. Management Science 2008;39(6):736–745.

[7] Lavoie S, Minoux M, Odier E. A new approach of crew pairing problems by column generation and application to air transport. European Journal of Operational Research 1988;35:45–58.

[8] Restrepo MI, Lozano L, Medaglia AL. Constrained network-based column generation for the multi-activity shift scheduling problem. International Journal of Production Economics 2012;140(1):466–472.

[9] Grönkvist M. Accelerating column generation for aircraft scheduling using constraint propagation. Computers and Operations Research 2006;33(10):2918–2934.

[10] Dumitrescu I, Boland N. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. Networks 2003;42(3):135–153.

[11] Zhu X, Wilhelm WE. A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation. Computers and Operations Research 2012;39(2):164–178.

[12] Santos L, Coutinho-Rodrigues J, Current JR. An improved solution algorithm for the constrained shortest path problem. Transportation Research Part B: Methodological 2007;41(7):756–771.

[13] Di L, Pugliese P, Guerriero F. Constrained shortest path problems 2012;1655(1980):1–19.

[14] Sedeño-Noda A, Alonso-Rodríguez S. An enhanced K-SP algorithm with pruning strategies to solve the constrained shortest path problem. Applied Mathematics and Computation 2015;265:602–618.

[15] Zeng W, Miwa T, Morikawa T. Application of the support vector machine and heuristic k-shortest path algorithm to determine the most eco-friendly path with a travel time constraint. Transportation Research Part D: Transport and Environment 2017;57:458–473.

[16] Lozano L, Medaglia AL. On an exact method for the constrained shortest path problem. Computers and Operations Research 2013;40(1):378–384.

[17] Thomas BW, Calogiuri T, Hewitt M. An exact bidirectional A* approach for solving resource-constrained shortest path problems. Networks 2019;73(2):187–205.

[18] Pohl I. Bi-directional and heuristic search in path problems. Stanford Linear Accelerator Center, Stanford University; 1969.

[19] Lozano L, Duque D, Medaglia AL. An exact algorithm for the elementary shortest path problem with resource constraints. Transportation science 2015;50(1):348–357.

[20] Duque D, Lozano L, Medaglia AL. An exact method for the biobjective shortest path problem for large-scale road networks. European Journal of Operational Research 2015;242(3):788–797.

[21] Bolívar MA, Lozano L, Medaglia AL. Acceleration strategies for the weight constrained shortest path problem with replenishment. Optimization Letters 2014;8(8):2155–2172.

[22] Duque D, Lozano L, Medaglia AL. Solving the orienteering problem with time windows via the pulse framework. Computers and Operations Research 2014;54:168–176.

[23] Duque D, Medaglia AL. An exact method for a class of robust shortest path problems with scenarios. Networks 2019;DOI:10.1002/net.21909.

[24] Lozano L, Smith JC. A backward sampling framework for interdiction problems with fortification. INFORMS Journal on Computing 2017;29(1):123–139.

[25] Arslan O, Jabali O, Laporte G. Exact solution of the evasive flow capturing problem. Operations Research 2018;66(6):1625–1640.

[26] Schrotenboer A, Ursavas E, Vis I. A branch-and-price-and-cut algorithm for resource constrained pickup and delivery problems. Transportation Science 2019;In press.

[27] Montoya A, Guéret C, Mendoza JE, Villegas JG. A multi-space sampling heuristic for the green vehicle routing problem. Transportation Research Part C: Emerging Technologies 2016;70:113–128.

[28] 9th DIMACS Implementation Challenge - Shortest Paths;. Accessed: 2019-07-08. http://users.diag.uniroma1.it/challenge9/.

[29] Dongarra JJ. Performance of various computers using standard linear equations software. ACM SIGARCH Computer Architecture News 2014;20(3):22–44.

[30] Bixby RE. Solving real-world linear programs: A decade and more of progress. Operations Research 2003;50(1):3–15.

[31] Quimper CG, Rousseau LM. A large neighbourhood search approach to the multi-activity shift scheduling problem. Journal of Heuristics 2010;16(3):373–392.

[32] Dumitrescu I, Boland N. Algorithms for the weight constrained shortest path problem. International Transactions in Operational Research 2001;8(1):15–29.

A | APPENDIX:MASSP AUXILIARY PROBLEM NETWORK

We transform problem (2) into a CSP with multiple resources on a directed acyclic network, in which nodes correspond to combinations of activities and time periods arranged by layers, where each layer corresponds to a time period.

Let s and e denote the start and end nodes, and let ordered pair $\{i, t\}$ denote a node corresponding to scheduling activity i at time t. Figure 11 presents the general structure of our network. We generate two types of arcs to denote the start of the shift (dashed arcs) and the assignment of an activity to a given time period (solid arcs). Node s is connected to all nodes except for e and nodes corresponding to a change of work activity (recalling that activity 1 is used to represent such changes). For $i \geq 2$ and $t < H$, nodes $\{i, t\}$ are connected to e, to node $\{i, t+1\}$, and to node $\{1, t+1\}$, representing the end of the shift, the continuation of activity i during time period $t+1$, or a change of activity in period $t+1$, respectively. Finally, for $2 \leq t \leq H-1$, nodes $\{1, t\}$ are connected to nodes $\{i, t+1\}$ having $i \geq 2$, which represents that a change of activity occurred during time t and a new activity is scheduled starting at time $t+1$.

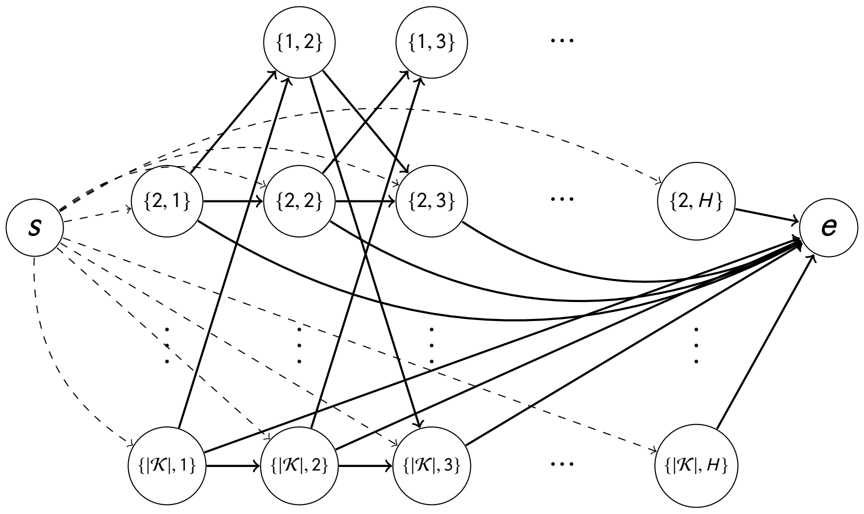


FIGURE 11 Auxiliary network for the pricing problem in a column generation approach for MASSP.

Every path in this network corresponds to a shift. Feasible shifts are obtained by enforcing the upper limits u_i on the number of time periods that an employee can perform each activity via resource constraints, where activity arcs consume one unit of a resource corresponding to the activity associated with the arc (see Restrepo et al. [8]).

B | APPENDIX:BIDIRECTIONAL PULSE STRATEGIES EXAMPLE

The purpose of this section is to show an example of how the bidirectional pulse strategies work in an intermediate stage of the algorithm. Figure 12 shows a network with six nodes and nine arcs. Each arc has an associated cost and resource consumption. In addition, the resource consumption limit is set to 15. Finally, the best solution found so far is $\mathcal{P}^* = \{s, v_2, v_3, e\}$ with a cost of 18 and a total resource consumption of 3. In the current stage of the algorithm, a pulse arriving from node s reaches node v_1 with a cost of 1, a resource consumption of 6, and a partial path $\mathcal{P} = \{s, v_1\}$ (see the red arc of the figure). Accordingly, before propagating this pulse to another node, the BP uses the pruning strategies to check if the current pulse can be pruned and if the best path can be updated.

First, we consider the bounds pruning strategy using the cost of the partial path, the cost of the minimum cost path from v_1 to e, and the primal bound. In this case the condition $1 + 5 < 18$ is true, so the pulse cannot be pruned by bounds pruning. Then, we consider the infeasibility pruning strategy using the resource consumption of the partial path, the resource consumption of the minimum resource consumption path from v_1 to e, and the resource consumption limit. In this case the condition $6 + 3 \leq 15$ is true, so the pulse cannot be pruned by this strategy. Finally, we consider the dominance pruning strategy. Note that the BP already found a path that reaches node v_1 on the forward direction and added the information to the non-dominated list of labels $\mathcal{L}^f(v_1)$. Consequently, we check if the partial path is dominated by the previously found path. In this case the partial path is not dominated by it, because even if the partial path has a higher resource consumption, it has a lower cost.

After checking the pruning strategies, the BP proceeds to check if the best solution can be updated using the path completion and the path joins strategies. Regarding the path completion strategy, we first check if the path can be completed using the minimum cost path. Formally we check if $1 + 5 < 18$ and $6 + 11 \leq 15$. In this case this path cannot be completed, as it is not feasible in terms of time. Then, we check if the path can be completed using the minimum resource consumption path. More specifically if $1 + 10 < 18$ and $6 + 3 \leq 15$. As both conditions are met, the path can be completed and the primal bound can be updated to 11. Finally, we check if we can join the path using partial paths created on the opposite direction. Formally, we check if $1 + 9 < 11$ and $6 + 7 \leq 15$. As both conditions are met, we join the partial paths and update the primal bound to 10.

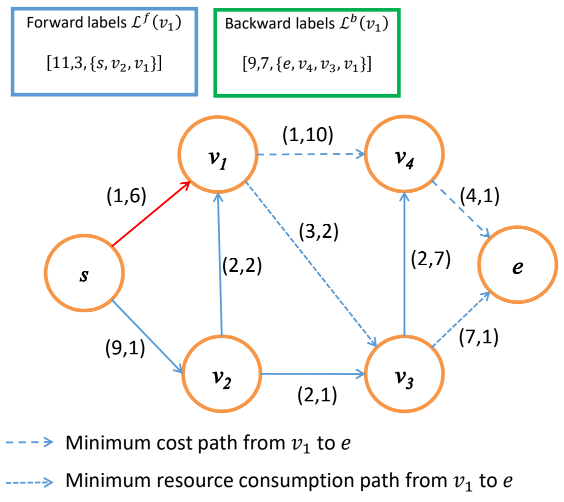

Forward labels $\mathcal{L}^f(v_1)$
$[11,3,\{s,v_2,v_1\}]$
Backward labels $\mathcal{L}^b(v_1)$
$[9,7,\{e,v_4,v_3,v_1\}]$
v_1
v_4
s
e
v_2
v_3
(1,10)
(1,6)
(4,1)
(3,2)
(2,2)
(2,7)
(9,1)
(7,1)
(2,1)
Minimum cost path from v_1 to e
Minimum resource consumption path from v_1 to e

FIGURE 12 A bidirectional pulse algorithm example