Computers & Operations Research 40 (2013) 378–384

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

On an exact method for the constrained shortest path problem

Leonardo Lozano, Andrés L. Medaglia *

Centro para la Optimización y Probabilidad Aplicada (COPA), Departamento de Ingeniería Industrial, Universidad de los Andes, Cr 1E No. 19A-10, ML711, Bogotá, Colombia

ARTICLE INFO

Available online 16 July 2012

Keywords:
Constrained shortest path
Shortest path problem
Column generation

ABSTRACT

The constrained shortest path (CSP) is a well known NP-Hard problem. Besides from its straightforward application as a network problem, the CSP is also used as a building block under column-generation solution methods for crew scheduling and crew rostering problems. We propose an exact solution method for the CSP capable of handling large-scale networks in a reasonable amount of time. We compared our approach with three different state-of-the-art algorithms for the CSP and found optimal solutions on networks with up to 40,000 nodes and 800,000 arcs. We extended the algorithm to effectively solve the auxiliary problems of a multi-activity shift scheduling problem and a bus rapid transit route design problem tackled with column generation. We obtained significant speedups against alternative column generation schemes that solve the auxiliary problem with state-of-the-art commercial (linear) optimizers. We also present a first parallel version of our algorithm that shows promising results.

1. Introduction

Consider a directed graph $\mathcal{G}=(\mathcal{N},\mathcal{A})$, where $\mathcal{N}=\{v_1,\ldots,v_i,\ldots,v_n\}$ represents the set of nodes and $\mathcal{A}=\{(i,j)|v_i\in\mathcal{N},\ v_j\in\mathcal{N},i\neq j\}$ the set of arcs. For each arc $(i,j)\in\mathcal{A}$ the nonnegative weights, c_{ij} and t_{ij}, represent the cost and resource consumption incurred by traversing the arc. The constrained shortest path (CSP) problem consists of finding a path $\mathcal{P}$ from a start node $v_s\in\mathcal{N}$ to an end node $v_e\in\mathcal{N}$ that minimizes the total cost, subject to not exceeding a maximum resource consumption T. The CSP is known to be NP-Hard even for the case of one resource [16].

Aside from its straightforward application (i.e., shortest path subject to a time limit), the CSP and its variants naturally arise as auxiliary problems in column generation schemes for air cargo planning and routing [11], flight planning [18], crew pairing [23,25], tail assignment problem in aircraft scheduling [20], day-to-day crew operation [32], and crew rostering problems [15], among others.

Some variants of the CSP include non-additive objective functions [27], probability functions defined over the network [2], forbidden paths [33], and replenishment arcs [30].

Solution strategies for the CSP can be classified into one of two main categories: Dynamic Programming (DP) and Lagrangian relaxation. Methods based on DP are also known as label-setting or label-correcting algorithms. Perhaps the most distinctive feature of these algorithms is that they can be extremely fast for reasonably sized networks; however, they might fail to scale well for very large networks due to the "curse of dimensionality" of DP, more specifically, the number of labels that need to be stored can be huge and, consequently, impossible to manage. One of the first algorithms in this category is the one proposed by Joksch [22], lately extended by Dumitrescu and Boland [13] by including preprocessing techniques. More recently, Zhu and Wilhelm [34,35] developed a three-stage approach specialized for column generation subproblems that transforms the CSP into a shortest path problem and solves it using a labeling method.

The second major solution approach for the CSP is based on the use of Lagrangian relaxation to solve the integer programming formulation of the problem. The efficiency of this approach relies on the effectiveness of the underlying unconstrained shortest path algorithms. Handler and Zang [21] solved the CSP by using a k-th shortest path algorithm; that is, they identify k paths, sort them by length, and evaluate them successively until they find the first path that satisfies the (side) constraint. The same authors also proposed a Lagrangian relaxation approach that can be seen as a generalization of the k-shortest path algorithm for the CSP, that is, a method that searches in a direction that (linearly) combines the information of the original objective and the (side) constraint. Santos et al. [29] extended this idea by refining the search direction based on the relative tightness of the (side) constraint. More recently, Carlyle et al. [5] used Lagrangian relaxation combined with enumeration of near-shortest paths.

This research aims to expand the body of knowledge of the CSP. Specifically, it proposes an exact algorithm that consistently

* Corresponding author. Tel.: +57 1 3394949x2880; fax: +57 1 3324321.
E-mail address: amedagli@uniandes.edu.co (A.L. Medaglia).
URL: http://wwwprof.uniandes.edu.co/~amedagli (A.L. Medaglia).

0305-0548/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2012.07.008

outperforms three state-of-the-art algorithms, namely, those by Santos et al. [29], Zhu and Wilhelm [35], and Dumitrescu and Boland [13], over a vast set of instances from the literature. In addition and most importantly, the proposed approach provides a flexible solution framework that can be easily parallelized and extended to include multiple side constraints and negative costs.

This paper is organized as follows. Section 2 presents a broad overview of the proposed *pulse algorithm*, focusing on its intuition. Section 3 provides detailed description of the algorithm's components. Section 4 presents our computational experience with the proposed algorithm against the state-of-the-art algorithms for the CSP. Section 5 illustrates our experience while embedding the algorithm within column generation schemes. Finally, Section 6 concludes the paper and outlines future extensions.

2. The pulse algorithm: intuition and overview

The idea behind the pulse algorithm is very simple, almost naive, yet very powerful. The algorithm is based on the idea of propagating pulses through a network from a start node $v_s \in \mathcal{N}$ to an end node $v_e \in \mathcal{N}$. As a pulse traverses the network from node to node, it builds a partial path $\mathcal{P}$ including the nodes already visited, the cumulative objective function $c(\mathcal{P})$ and the cumulative resource consumption $t(\mathcal{P})$. Each pulse that reaches the final node v_e contains all the information for a feasible path $\mathcal{P}$ from v_s to v_e. If nothing prevents the pulses from propagating, the algorithm completely enumerates all possible paths from v_s to v_e, ensuring that the optimal path $\mathcal{P}^*$ is always found. At the core of the algorithm lies the ability to (effectively and aggressively) prune pulses (i.e., prevent their propagation), without jeopardizing the optimal path. This idea is shared with other algorithms like branch and bound, where an implicit enumeration is performed with relative efficiency. Similarly, the strength of the pulse algorithm depends on the pruning strategies; in our case for the CSP, pruning by *dominance*, *bounds*, and *infeasibility*.

Algorithm 1 presents the pseudocode of the pulse algorithm. Lines 1 through 3 initialize the partial path $\mathcal{P}$, the initial objective function c^0, and the initial resource consumption t^0. Line 4 runs a one-to-all shortest path algorithm to find the minimum resource consumption for every node (see Section 3.2). Line 5 invokes the recursive function `pulse` starting the propagation from node v_s. Line 6 returns the optimal path $\mathcal{P}^*$ found in the recursion.

Algorithm 1. Pulse algorithm.

Input: $\mathcal{G}$, v_s, v_e, T
Output: Optimal path $\mathcal{P}^*$
1: $\mathcal{P} \leftarrow \{\}$
2: $c^0 \leftarrow 0$
3: $t^0 \leftarrow 0$
4: `initialization`($\mathcal{G}$,T) {*see Section* 3.3}
5: `pulse`(v_s,c^0,t^0,$\mathcal{P}$)
6: **return** $\mathcal{P}^*$

Algorithm 2 shows the body of the recursive function `pulse`, where $\mathcal{N}(v_k) = \{v_i \in \mathcal{N} | (k,i) \in \mathcal{A}\}$ is the set of head nodes of the outgoing arcs from node v_k. Each time the function `pulse` is invoked on the final node v_e, the information for the best path known is stored globally and the pulse is no longer propagated. Lines 1 through 3 use the different pruning strategies in order to prune the incoming pulse. If the pulse is not pruned, line 4 adds the current node to the partial path and lines 5 through 9 propagate the pulse by recursively invoking the function `pulse` on all nodes $v_i \in \mathcal{N}(v_k)$.

Algorithm 2. `Pulse` function.

Input: v_k, c, t, $\mathcal{P}$
Output: `void`
1: **if** `checkDominance`(v_k,c,t) = `false` **then** {*see Section* 3.1}
2: **if** `checkFeasibility`(v_k,t) = `true` **then** {*see Section* 3.2}
3: **if** `checkBounds`(v_k,c) = `false` **then** {*see Section* 3.3}
4: $\mathcal{P}' \leftarrow \mathcal{P} \cup \{v_k\}$
5: **for** $v_i \in \mathcal{N}(v_k)$ **do**
6: $c' \leftarrow c + c_{ki}$
7: $t' \leftarrow t + t_{ki}$
8: `pulse`(v_i,c',t',$\mathcal{P}'$)
9: **end for**
10: **end if**
11: **end if**
12: **end if**

Note that the way the pulse algorithm explores the graph follows a depth-first search (DFS) exploration scheme truncated by the pruning strategies. An illustrative animation of the algorithm is available at http://hdl.handle.net/1992/1144.

3. Pruning strategies

The pulse algorithm heavily depends on the strength of its pruning strategies. This section outlines the different strategies implemented for the CSP. It is noteworthy that given that all weights defined over the arcs are nonnegative, the optimal path must be an elementary path that contains no cycles. Whenever the function `pulse` is invoked on a node v_k, the algorithm verifies if visiting node v_k creates a cycle in the partial path $\mathcal{P}$ associated with the pulse. This verification is performed in constant time using a binary vector that keeps a record of the nodes already visited. Paths containing cycles are not considered in the search.

3.1. Pruning by dominance

Dominance relationships can be defined over partial paths as done in DP-based algorithms that extend Bellman-Ford's cf. [24]. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two partial paths to a given node v_k. If $c(\mathcal{P}_1) > c(\mathcal{P}_2)$ and $t(\mathcal{P}_1) > t(\mathcal{P}_2)$ then $\mathcal{P}_1$ is said to be strongly dominated by $\mathcal{P}_2$. If $c(\mathcal{P}_1) > c(\mathcal{P}_2)$ and $t(\mathcal{P}_1) \geq t(\mathcal{P}_2)$ or $c(\mathcal{P}_1) \geq c(\mathcal{P}_2)$ and $t(\mathcal{P}_1) > t(\mathcal{P}_2)$ then $\mathcal{P}_1$ is said to be (weakly) dominated by $\mathcal{P}_2$. If $c(\mathcal{P}_1) = c(\mathcal{P}_2)$, $t(\mathcal{P}_1) = t(\mathcal{P}_2)$, and $\mathcal{P}_1 \neq \mathcal{P}_2$ then $\mathcal{P}_1$ and $\mathcal{P}_2$ are alternative paths to node v_k.

Given these relations, we use labels to check for strong dominance, (weak) dominance, or the existence of alternative paths. For each node $v_k \in \mathcal{N}$ we define a limited *memory* $\mathcal{L}(v_k) = \{(c_{kl},t_{kl}) | l = 1,\ldots,Q\}$ where c_{kl} and t_{kl} are the cost and resource consumption labels, respectively; and Q denotes the total number of labels (i.e., memory size). At first glance, it seems reasonable to store all the values for $c(\mathcal{P})$ and $t(\mathcal{P})$ corresponding to the paths that have passed through a given node v_k, so that it is possible to prune any incoming pulse in case it is dominated. However, this exhaustive approach would require huge amounts of memory to store the values for all partial paths in large networks, ultimately, forcing the method to fail in terms of scalability. In contrast, by choosing a limited memory of Q labels (per node), it is possible that some dominated paths may inefficiently pass through node v_k, but still the optimal solution is never pruned (just dominated paths).

In contrast to a classical DP approach, it is noteworthy that these labels are not constantly expanding, nor the optimality

depends on storing them all. We evaluated the impact of different storage rules and values of Q on the computational performance of the pulse algorithm. We tried the following storage rules: (1) to randomly replace a pair of labels each time a new nondominated path is found (*random rule*); (2) to always keep the labels with the lowest cost and resource consumption (*elitist rule*); and (3) to use a distance metric to rank the labels and keep always a diversified (heterogeneous) label set (*diversified rule*). For the latter rule, we have borrowed the crowding distance from NSGA-II [9], where it is used as a parameter-less approach to preserve a diversified population in the approximate Pareto frontier. We defer until Section 4 the discussion regarding the computational efficiency of these rules. In summary, these labels prune dominated partial paths, yet they consume a reasonable (and limited) amount of memory.

3.2. Pruning by infeasibility

The idea of this strategy is to prune pulses at an early stage, as soon as it is known that the traveling pulse will not be able to reach the final node meeting the (side) constraint. To do so, it is necessary to calculate the minimum resource consumption from any node v_k to the final node v_e. To obtain this information we reverse the original network (i.e., change the direction of all arcs) creating a new directed graph $\mathcal{G}'=(\mathcal{N},\mathcal{A}')$ where $\mathcal{A}'=\{(j,i)|(i,j)\in\mathcal{A}\}$ and the weights are set to $c'_{ji}=c_{ij}$ and $t'_{ji}=t_{ij}$ for $(j,i)\in\mathcal{A}'$. The new start node is the original end node $v'_s=v_e$. Next, we run a one-to-all shortest path algorithm to find the minimum resource consumption from the initial node to all other nodes in graph $\mathcal{G}'$. This is equivalent to find the minimum resource consumption $\underline{t}(v_k)$ from any node v_k to the final node v_e in the original network. Then, we set the maximum resource consumption $\overline{t}(v_k)$ for a partial path up to any given node v_k to $\overline{t}(v_k)=T-\underline{t}(v_k)$. That is, if for a partial path $\mathcal{P}$ to node v_k it is known that $t(\mathcal{P})>\overline{t}(v_k)$, then partial path $\mathcal{P}$ is infeasible because $t(\mathcal{P})+\underline{t}(v_k)>T$. The procedure to find all the $\overline{t}(v_k)$ values is done as part of the initialization of the algorithm.

This pruning strategy shares the same spirit of similar algorithms that have been used to reduce the size of the original network, but it does it in a much simpler way. In contrast to the preprocessing phases of the algorithms by Dumitrescu and Boland [13] and Zhu and Wilhelm [35], where they use iterative procedures that calculate forward and backward shortest paths for all nodes and modify the original graph by deleting nodes and arcs, we only use a backward one-to-all shortest path algorithm for obtaining the lower bounds on the resource consumption. Gromicho et al. [19] also propose a similar idea to restrict the state space in a DP context by using a feasibility check before extending the labels, but without the look-ahead component of our lower bounds.

The strength of this pruning strategy relies on the fact that it prunes infeasible partial paths at early stages of the exploration. For problems where the resource constraint is very tight, the algorithm takes advantage of this (early) aggressive pruning to explore the network fast, yet ensuring that the optimal solution is never pruned (only infeasible paths).

3.3. Pruning by bounds

This pruning strategy is inspired on the power of primal and dual bounds in branch and bound. These bounds store the best objective function found (primal bound) and the most optimistic information made available at the unexplored nodes (dual bound).

Every time a pulse reaches the last node v_e, a feasible path is generated and the primal bound $\overline{c}$ may be updated with the best known objective function value. Given that all costs are nonnegative, if a partial path $\mathcal{P}$ exceeds the primal bound, that is $c(\mathcal{P})\geq\overline{c}$, this path can be safely pruned because another path with a lower (or equal) total cost has already been found. Furthermore, we derived a bound on the minimum cost $\underline{c}(v_k)$ from any node v_k to the final node v_e, following a similar procedure to the one used for $\underline{t}(v_k)$. Again, it is necessary to reverse the network and use a one-to-all shortest path algorithm to minimize the cost from the first node $v'_s=v_e$ in the reversed network to any other node v_k; this is equivalent to find the minimum cost $\underline{c}(v_k)$ from any node v_k to v_e in the original network. As this shortest path algorithm does not take into account the resource consumption, the values $\underline{c}(v_k)$ are a dual bound on the real minimum cost subject to the resource constraint. Given a partial path $\mathcal{P}$ to node v_k, if $c(\mathcal{P})+\underline{c}(v_k)\geq\overline{c}$, then path $\mathcal{P}$ can be safely pruned because the algorithm has already found a complete feasible path that is better (or equal) than any complete path that includes the partial path $\mathcal{P}$. The procedure to find all the dual bounds $\underline{c}(v_k)$ is done as part of the initialization of the algorithm. The use of the dual bound strengthens the algorithm at the expense of a single run of a one-to-all shortest path algorithm. It is worth mentioning, that similar bounds have been used by A^*-based shortest path algorithms [26].

4. Computational experiments for the CSP

In this section we compare the pulse algorithm against the algorithms by Santos et al. [29], Zhu and Wilhelm [35], and Dumitrescu and Boland [13] on a wide range of benchmark problems from the literature. The proposed algorithm was implemented in Java, compiled using Eclipse SDK version 3.4.2, and the experiments executed on a computer with an Intel Mobile Core 2 Duo @ 2.4 GHz CPU P8600 with 512 MB of RAM allocated to the memory heap size of the Java Virtual Machine on Windows XP Professional. The shortest path problems were solved using our Dijkstra's double buckets implementation adapted from Cherkassky et al. [6].

Regarding the pruning by dominance strategy, a mixture between the random and elitist storage rules (with three labels) was proven to be both fast and effective. At each node we keep a first nondominated label that is overwritten every time that a partial path exhibits a lower cost than the one stored, a second nondominated label that is overwritten every time that a partial path exhibits a lower resource consumption than the one stored, and a third nondominated label that is replaced randomly.

4.1. Experiments on large-scale instances (with one side constraint)

We used the exact same benchmark problems used by Santos et al. [29]; a testbed that comprises 180 instances organized in three groups. The first group contains the smaller instances with $|\mathcal{N}|=10{,}000$ and $|\mathcal{A}|=15{,}000$, 25,000, 50,000, 100,000, 150,000, and 200,000; the second group contains the middle-sized instances with $|\mathcal{N}|=20{,}000$ and $|\mathcal{A}|=30{,}000$, 50,000, 100,000, 200,000, 300,000, and 400,000; and the third group contains the larger instances with $|\mathcal{N}|=40{,}000$ and $|\mathcal{A}|=60{,}000$, 100,000, 200,000, 400,000, 600,000, and 800,000. For each $|\mathcal{N}|-|\mathcal{A}|$ combination in each group, there are 10 different instances. Furthermore, Santos et al. [29] define the maximum resource consumption T based on the tightness of the constraint denoted by p and defined as follows:

$$p=\frac{T-t(\mathcal{P}_t^*)}{t(\mathcal{P}_c^*)-t(\mathcal{P}_t^*)}$$

where $\mathcal{P}_c^*$ and $\mathcal{P}_t^*$ are the paths that minimize cost and resource consumption, respectively; and $t(\mathcal{P})$ is the resource consumption

Table 1
Computational results for the Santos et al. [29] instances.

Nodes	Arcs	$p=0.1$			$p=0.2$			$p=0.4$			$p=0.6$			$p=0.8$		
		Pulse	LRA	Speedup	Pulse	LRA	Speedup	Pulse	LRA	Speedup	Pulse	LRA	Speedup	Pulse	LRA	Speedup
10,000	15,000	0.01	0.60	60.00	0.01	0.60	60.00	0.01	0.60	60.00	0.01	0.60	60.00	0.01	0.60	60.00
10,000	25,000	0.02	0.70	35.00	0.02	0.70	35.00	0.02	0.70	35.00	0.02	0.70	35.00	0.02	0.70	35.00
10,000	50,000	0.02	1.10	55.00	0.02	1.10	55.00	0.02	1.10	55.00	0.02	1.10	55.00	0.02	1.10	55.00
10,000	100,000	0.04	1.80	45.00	0.05	1.80	36.00	0.04	1.80	45.00	0.05	1.80	36.00	0.04	1.80	45.00
10,000	150,000	0.07	2.50	35.71	0.06	2.50	41.67	0.07	2.50	35.71	0.07	2.60	37.14	0.06	2.80	46.67
10,000	200,000	0.07	2.70	38.57	0.14	2.70	19.29	0.16	2.70	16.88	0.10	2.80	28.00	0.08	2.80	35.00
20,000	30,000	0.02	1.20	60.00	0.02	1.20	60.00	0.02	1.20	60.00	0.02	1.20	60.00	0.02	1.20	60.00
20,000	50,000	0.03	1.50	50.00	0.03	1.50	50.00	0.03	1.50	50.00	0.03	1.50	50.00	0.03	1.50	50.00
20,000	100,000	0.06	2.30	38.33	0.06	2.30	38.33	0.06	2.30	38.33	0.06	2.30	38.33	0.06	2.40	40.00
20,000	200,000	0.11	3.70	33.64	0.09	3.70	41.11	0.09	3.70	41.11	0.09	3.80	42.22	0.12	3.90	32.50
20,000	300,000	0.17	4.80	28.24	0.14	4.80	34.29	0.15	4.90	32.67	0.15	4.90	32.67	0.19	5.00	26.32
20,000	400,000	0.17	5.90	34.71	0.22	5.90	26.82	0.28	5.90	21.07	0.23	6.00	26.09	0.18	6.20	34.44
40,000	60,000	0.05	2.40	48.00	0.05	2.40	48.00	0.05	2.40	48.00	0.05	2.40	48.00	0.05	2.40	48.00
40,000	100,000	0.08	3.00	37.50	0.08	3.00	37.50	0.08	3.00	37.50	0.08	3.10	38.75	0.08	3.10	38.75
40,000	200,000	0.12	4.90	40.83	0.12	4.90	40.83	0.12	4.90	40.83	0.12	5.00	41.67	0.12	5.10	42.50
40,000	400,000	0.21	7.70	36.67	0.22	7.70	35.00	0.26	7.80	30.00	0.22	8.00	36.36	0.21	8.30	39.52
40,000	600,000	0.29	10.70	36.90	0.34	10.70	31.47	0.31	10.90	35.16	0.31	11.20	36.13	0.31	11.70	37.74
40,000	800,000	0.39	13.00	33.33	0.47	13.10	27.87	0.52	13.40	25.77	0.55	13.80	25.09	0.43	14.40	33.49
Geometric mean				40.32			38.41			37.26			41.22			41.10

for a given path $\mathcal{P}$. Each one of the 180 instances was solved setting $p=0.1$, 0.2, 0.4, 0.6, and 0.8, thus leading to a total of 900 runs of the pulse algorithm. Note that $p \in [0,1]$; and low (high) values of p mean that the resource consumption constraint is tight (loose).

Table 1 compares the results of the pulse algorithm against the Lagrangian relaxation algorithm (LRA) proposed by Santos et al. [29] over different values for the tightness of the resource constraint (namely, parameter p). The instances are organized by rows in three categories (i.e., small, medium, and large-sized instances) based on the number of nodes and arcs (columns 1 and 2). Columns 3–7 show, for the different values of p, the average execution time (in seconds) for the pulse algorithm, the average execution time as reported by Santos et al. [29], and the speedup calculated as the ratio between the execution times. Finally, the last row reports the geometric mean of the speedups. Note that the geometric mean, opposed to the arithmetic mean, avoids being overly optimistic with large ratios obtained on few instance sizes (node-arc combination), thus it provides a fairer comparison of speedups [3]. For the sake of fairness we scaled our computation times using the LINPACK benchmark [12]. According to this benchmark, our computer is approximately 2.1 times faster than the one used by Santos et al. [29], so all the reported times were scaled by this factor.

The results presented in Table 1 show that the pulse algorithm consistently outperforms LRA in every instance, regardless of the constraint tightness. Speedups range from 16 to 60 while the geometric mean shows that the pulse algorithm is consistently roughly 40 times faster than LRA. In terms of absolute computing time (not scaled), the pulse algorithm solved all the small networks in less than 0.1 s, the middle-sized networks in less than 0.15 s, and the large networks in less than 0.3 s. Profiling our algorithm we found that most of the computational time for these instances is spent at the initialization.

4.2. Experiments on small and mid-sized instances (with multiple side constraints)

Zhu and Wilhelm [35] present two sets of instances adapted from Beasley and Christofides [1] that range in size from 100 nodes and 959 arcs to 500 nodes and 4978 arcs. The problems in the first set consider one resource constraint while those in the second set consider 10 resource constraints. For each problem in Beasley and Christofides [1], Zhu and Wilhelm [35] build 100 random instances with different arc costs, for a total of 600 instances per set. Zhu and Wilhelm [35] extended the label-setting algorithm (LSA) of Dumitrescu and Boland [13] to handle multiple resources and used it as benchmark algorithm for their own three-stage approach (TSA). Given that TSA is intended for solving multiple subproblems under a column generation scheme, Zhu and Wilhelm [35] define a *threshold* as the number of instances (i.e., subproblems) solved with the benchmark algorithm required to match the total time spent by TSA, including its preprocessing procedures. If under a column generation scheme, it happens that the number of subproblems to be solved is larger than the threshold, it means that it is worth using TSA over the benchmark algorithm. Zhu and Wilhelm [35] define the threshold as follows:

$$\tau_{(\mathrm{TSA},\cdot)} = \begin{cases} \left\lceil \dfrac{t^1_{\mathrm{TSA}} + t^2_{\mathrm{TSA}}}{\bar{t}_{\cdot} - \bar{t}^3_{\mathrm{TSA}}} \right\rceil & \text{if } \bar{t}_{\cdot} > \bar{t}^3_{\mathrm{TSA}} \\ \infty & \text{if } \bar{t}_{\cdot} \le \bar{t}^3_{\mathrm{TSA}} \end{cases}$$

where t^1_{TSA} and t^2_{TSA} are the times it takes TSA to complete stages 1 and 2 of the preprocessing procedure at the beginning of the column generation scheme; and $\bar{t}^3_{\mathrm{TSA}}$ and $\bar{t}_{\cdot}$ are the average times it takes a single run of the stage 3 of TSA and the benchmark algorithm $(\cdot)$. Note that if the average time of the benchmark algorithm $(\cdot)$ is less than the average time of stage 3 of TSA, it is said that the benchmark algorithm outperforms TSA regardless of the number of subproblems to be solved, thus the threshold $\tau_{(\mathrm{TSA},\cdot)} \triangleq \infty$.

Table 2 compares the pulse algorithm with LSA and TSA on the exact same set of instances by Zhu and Wilhelm [35]. We extended the pulse algorithm to handle multiple resources but to keep the computational budget under control, we obtained lower bounds with the one-to-all shortest path for the cost and just one resource, while for the other resources the lower bounds were set to zero. Columns 1–4 show the instance, number of resource constraints, number of nodes, and number of arcs, respectively. Columns 5 and 6 present the average execution time to solve each subset of 100 instances for LSA and for the pulse algorithm. Column 7 exhibits the speedup achieved with

Table 2
Computational results for the Zhu and Wilhelm [35] instances.

Instance	Resources	Nodes	Arcs	LSA (s)	Pulse (s)	Speedup	TSA (s)		$\tau_{(TSA,Pulse)}$	$\tau_{(TSA,LSA)}$
							$t^1_{TSA}+t^2_{TSA}$	$\bar{t}^3_{TSA}$		
rcsp3	1	100	959	0.005	0.00067	7.5	0.140	0.00016	274.5	30
rcsp4	1	100	959	0.005	0.00069	7.3	0.125	0.00016	236.5	25
rcsp11	1	200	1971	0.004	0.00073	5.5	0.625	0.00093	∞	222
rcsp12	1	200	1971	0.006	0.00072	8.3	0.516	0.00094	∞	100
rcsp19	1	500	4978	0.006	0.00112	5.3	1.094	0.00297	∞	411
rcsp20	1	500	4978	0.006	0.00119	5.1	0.891	0.00187	∞	203
rcsp7	10	100	999	0.198	0.00443	44.7	0.906	0.00000	204.6	5
rcsp8	10	100	999	0.085	0.00418	20.4	0.188	0.00016	46.8	2
rcsp15	10	200	1960	0.513	0.00654	78.5	1.594	0.00000	243.9	3
rcsp16	10	200	1960	0.192	0.00459	41.8	0.313	0.00000	68.2	2
rcsp23	10	500	4868	6.224	0.00670	928.9	99.658	0.00109	17,763.1	16
rcsp24	10	500	4868	2.975	0.01361	218.6	7.859	0.00000	577.4	3

the pulse algorithm, that is, how many times faster is the pulse over LSA. Columns 8 and 9 present the time used in preprocessing and stage 3 for TSA. Column 10 shows the threshold that compares the pulse algorithm with TSA, namely, the number of runs (subproblems) that could be solved with the pulse, before it starts to pay off to use TSA. Finally, column 11 shows the threshold that compares LSA with TSA. Zhu and Wilhelm [35] coded their algorithms in C/C++ and executed their experiments on an Intel 3.2 GHz dual core CPU with 2 GB of RAM. Given that their computer is better than ours, we assumed this handicap by not scaling our times.

For the first set of instances (with one resource constraint), Table 2 shows that the pulse algorithm consistently outperforms LSA in all the experiments, with speedups ranging from 5.1 to 8.3. The pulse algorithm dominated TSA in 4 out of 6 experiments (denoted by the threshold of ∞), meaning that regardless of the number of subproblems solved, the total execution time of the pulse algorithm will always be less than that of TSA. For those two instances without proof of dominance by the pulse, the threshold value gives the pulse a comfortable margin of at least 236 subproblems. In terms of absolute computing time, all the experiments were solved under 0.0012 s in average. For the second set of instances (with 10 resource constraints) the pulse algorithm significantly outperforms LSA reaching speedups of up to 900 times in the largest instances. It also compares favorably with TSA, achieving threshold values of up to 17,700 and solving all the experiments under 0.014 s. Over both sets of instances, the threshold value obtained with the pulse is several orders of magnitude larger than the one obtained with LSA.

4.3. Parallelizing the pulse algorithm

Given that the recursive `pulse` function explores one node at a time until it reaches the end node, it is possible to call the function over different nodes in parallel threads taking special care on how to update the global information. We parallelized the execution of the pulse algorithm by triggering at node v_s different threads that begin the exploration starting from the outgoing arcs. Two different threads cannot execute the same functions over the same node at the same time, if this happens, one thread must wait for the other to finish. In order to test our approach, we used the hardest instance (40,000 nodes and 800,000 arcs) from Santos et al. [29] and derived 10 new instances by randomly replacing the end node. Table 3 shows the execution time in seconds required to solve the instances with 1, 2, 4, and 8 threads. The last column shows the speedup calculated as the ratio between the best multi-thread approach and the single-thread approach. For this experiment we used a machine with two processors Intel Xeon X5450 running at 3 GHz, 8 GB of RAM, and a 64-bit Windows Vista Ultimate operating system.

Table 3
Comparing the execution time (s) for the single-thread and the multi-thread versions of the pulse algorithm on instances with 40,000 nodes and 800,000 arcs.

Instance	Threads				Speedup
	1	2	4	8	
1	0.384	0.279	0.208	0.194	1.98
2	0.237	0.204	0.195	0.149	1.59
3	0.221	0.185	0.161	0.143	1.55
4	0.192	0.202	0.138	0.170	1.39
5	0.182	0.163	0.161	0.168	1.13
6	0.196	0.146	0.183	0.190	1.34
7	0.170	0.171	0.152	0.182	1.12
8	0.208	0.224	0.148	0.140	1.48
9	0.228	0.208	0.142	0.137	1.66
10	0.284	0.255	0.170	0.166	1.71

Table 3 shows that the parallel version of the pulse algorithm is faster than the single-thread version for all the tested instances. The speedups range from 1.12 to 1.98. These promising speedups might be the result of obtaining tighter primal bounds earlier by exploring different regions of the network at the same time in different threads. However, further experimentation must be made in order to fully exploit and understand the advantages and tradeoffs while using the parallel version of the algorithm.

5. Embedding the pulse algorithm within column generation schemes

Our experience has shown that an effective and practical use of the pulse algorithm arises while solving auxiliary problems in column generation procedures. Note, however, that it is often the case that these auxiliary problems contain negative costs generated from the dual variables of the master problem. We successfully adapted the pulse algorithm to solve the auxiliary problems for a Multi-Activity Shift Scheduling Problem (MASSP) and a Bus Rapid Transit Route Design Problem (BRTRDP). The computational experiments in this section were executed on a Dell Precision 7400 with 8 GB of RAM, two processors Intel Xeon X5450 running at 3 GHz, on a 64-bit Windows Vista Ultimate operating system.

5.1. Multi-activity shift scheduling problem

The Shift Scheduling Problem (SSP) deals with the selection of a set of shifts to satisfy a daily demand for staff requirements. The

Multi-Activity Shift Scheduling Problem (MASSP) is an extension of the SSP where the staff may perform several work activities in the same shift and the assignment of activities to a shift is restricted by work rules [8,7,10]. Depending on the problem at hand, these work rules include activity length, number of breaks, shift starting time, shift length, and maximum number of activities per shift, among others. Typically, in the MASSP the schedule spans over a multi-day planning horizon.

Under a column generation scheme for the MASSP we modeled the auxiliary problem as a Shortest Path Problem with Resource Constraints (SPPRC) with local time windows over each node and global resource constraints [28]. This variant of the SPPRC is defined over an acyclic graph with negative costs, several problem-specific side constraints, and up to 2000 nodes and 50,000 arcs. To solve the SPPRC, we modified the pulse algorithm as follows: being the graph acyclic, we did not have to check for cycles; the dominance pruning strategy cannot prune partial paths that have not met the minimum requirements for all resources; the infeasibility pruning strategy was extended to prune using multiple resource constraints as in Section 4.2; and the bounds pruning strategy still applies. Table 4 compares the performance on three real-world MASSP instances solved using traditional column generation, where the auxiliary problem is solved using integer programming, against a pulse-embedded column generation. The first column shows the instance name where A, TIL and D are the prefixes for the number of activities, the time interval length in minutes, and the planning horizon in days. Columns 2 and 3 show the total computational time needed using integer programming (IP) and the pulse algorithm for the auxiliary problems. All the integer programs were solved using Xpress-MP optimizer version 19.00.00.

Adapting the pulse algorithm to address this SPPRC (arising from the MASSP) was a relatively easy task given that the search logic of the algorithm remains unchanged and the multiple work rules for the shifts made the infeasibility pruning strategy very strong. The faster computational times allowed us to solve harder and larger instances that were simply not tractable with one of the best branch and bound implementations found in commercial optimizers (i.e., Xpress-MP).

5.2. Bus rapid transit route design problem

The Bus Rapid Transit Route Design Problem (BRTRDP) is the problem of finding a set of routes and frequencies that minimizes the operational and the passenger costs (travel time) while simultaneously satisfying the system's technical constraints, such as meeting the demands for trips, bus frequencies, and lane capacities.

To solve the BRTRDP we used a decomposition approach where new routes are added dynamically to a master problem [14,17]. Similar to Section 5.1, an auxiliary problem searches a network for suitable routes to be added to the master problem. In contrast to the SPPRC (in the MASSP) this network contains negative costs and possibly negative cost cycles. Given the physical infrastructure, there is a constraint that forbids a route to go backwards, meaning that buses stop in stations in a given direction (between terminal stations). To address this auxiliary problem, we modified the pulse algorithm as follows: the dominance pruning strategy remains unchanged; the infeasibility pruning strategy discards any path that visits a forbidden station; and the bounds pruning strategy is no longer used because of the negative cost cycles. Table 5 compares two column generation schemes for the BRTRDP: one that solves the auxiliary problem as an integer program using Xpress-MP optimizer version 19.00.00; and a second one that uses the pulse algorithm. The first column shows the instance name where S is the prefix for the number of stations in the system. Columns 2 and 3 show the total computational time needed for both approaches.

The computational speedup is up to 379 times due to the pulse and it is noteworthy that larger instances show the most significant improvement. Although the bound pruning strategy was not used, the remaining strategies were strong enough to achieve these results. The faster computational times allowed us to solve instances that the approach with integer programming was simply unable to solve.

Table 4
Speeding up the solution of real-world MASSP instances with the pulse algorithm.

Instance	Time (s)	
	IP[a]	Pulse
A5-TIL30-D7	3549	17
A7-TIL30-D7	71,216	83
A16-TIL30-D7	848,402	629

[a] Xpress-MP optimizer version 19.00.00.

Table 5
Speeding up the solution of the BRTRDP with the pulse algorithm.

Instance	Time (s)	
	IP[a]	Pulse
BRT-S8	9	2
BRT-S10	39	7
BRT-S15	2566	63
BRT-S19	140,369	370

[a] Xpress-MP optimizer version 19.00.00.

6. Conclusions and future work

We introduced an exact algorithm for the CSP that solves large networks with up to 40,000 nodes and 800,000 in a reasonable amount of time (under 0.3 s). The algorithm, based on the idea of propagating pulses through a network, performs better than the algorithm by Santos et al. [29] over a large testbed from the literature, achieving speedups of up to 60 times. It also outperforms the label-setting algorithm by Dumitrescu and Boland [13] reaching speedups of up to 900 times on the proposed instances and compares favorably with the up-to-date approach by Zhu and Wilhelm [35].

Additionally, this algorithm can be seen as a flexible framework able to handle difficult constraints modeled as pruning strategies. The idea of the algorithm is easy to understand and to implement, and the pruning strategies can be used as modules and adapted to problem specific conditions. Also, the only parameters to be defined by the user are the maximum number of labels stored at each node (i.e., Q), and the label storage rule.

We show that the algorithm is easy to parallelize, and illustrate this fact with a first parallel version of the algorithm that exhibits promising results when compared with the single-thread version of the algorithm on large-scale networks.

From a practical perspective, the pulse algorithm has been successfully extended for solving auxiliary problems under a column generation scheme arising in two different contexts: a bus rapid transit route design problem and a multi-activity shift scheduling problem. A very similar auxiliary problem arises in the context of the integrated vehicle- and crew-scheduling problem [31], where the pulse algorithm is likely to accelerate the column generation process.

Work currently underway includes: extending the algorithm to tackle the Elementary Shortest Path Problem with Resource Constraints (ESPPRC), a problem that arises while solving the Vehicle Routing Problem with Time Windows (VRPTW) under a column-generation procedure; applying the algorithm to a real medical task scheduling problem; extending its functionality to tackle a subproblem in the public transport line planning problem [4]; extending the pulse algorithm to the biobjective shortest path problem; and exploring further the parallelization of the algorithm to take full advantage of multiple core processors or graphics processing units.

Acknowledgments

The authors would like to thank Professors João Coutinho-Rodrigues at Universidade de Coimbra (Portugal), Xiaoyan Zhu at the University of Tennessee (U.S.A), and Wilbert E. Wilhelm at Texas A&M University (U.S.A) for their generosity in sharing with us their testbeds for the CSP. Professors Louis-Martin Rousseau at École Polytechnique de Montréal and Juan G. Villegas at Universidad de Antioquia for their insightful comments. Last, but not least, we want to thank Daniel Duque, our MS student at Universidad de los Andes, who developed the double-bucket shortest path implementation and coded an animation of the pulse algorithm that facilitates its understanding.

References

[1] Beasley JE, Christofides N. An algorithm for the resource constrained shortest path problem. Networks 1989;19:379–94.
[2] Bertsekas DP, Tsitsiklis JN. An analysis of stochastic shortest path problems. Mathematics of Operations Research 1991;16(3):580–95.
[3] Bixby RE. Solving real-world linear programs: a decade and more of progress. Operations Research 2002;50(1):3–15.
[4] Borndörfer R, Grötschel M, Pfetsch ME. A column-generation approach to line planning in public transport. Transportation Science 2007;41(1):123–32.
[5] Carlyle WM, Royset JO, Wood RK. Lagrangian relaxation and enumeration for solving constrained shortest-path problems. Networks 2008;52(4):256–70.
[6] Cherkassky BV, Goldbberg AV, Radzik T. Shortest path algorithms: theory and experimental evaluation. Mathematical Programming 1996;73:129–74.
[7] Côté M-C, Gendron B, Quimper C-G, Rousseau L-M. Formal languages for integer programming modeling of shift scheduling problems. Constraints 2009;16(1):54–76.
[8] Côté M-C, Gendron B, Rousseau L-M. Grammar-based integer programming models for multi-activity shift scheduling. Management Science 2011;57(1):1–13.
[9] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multi-objective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 2002;6(2):182–97.
[10] Demassey S, Pesant G, Rousseau L-M. A cost-regular based hybrid column generation approach. Constraints 2006;11(4):315–33.
[11] Derigs U, Friederichs S, Schäfer S. A new approach for air cargo network planning. Transportation Science 2009;43(3):370–80.
[12] Dongarra JJ. Performance of various computers using standard linear equations software. Technical report CS-89-85, University of Tennessee, USA; 2009.
[13] Dumitrescu I, Boland N. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. Networks 2003;42(3):135–53.
[14] Feillet D, Gendreau M, Medaglia AL, Walteros JL. A note on branch-and-cut-and-price. Operations Research Letters 2010;38(5):346–53.
[15] Gamache M, Soumis F, Marquis M, Desrosiers J. A column generation approach for large-scale aircrew rostering problems. Operations Research 1999;47(2):247–63.
[16] Garey MR, Johnson DS. Computers and intractability: a guide to the theory of NP-completeness. New York: W.H. Freeman; 1979.
[17] González JE, Lozano L, Walteros JL, Feillet D, Medaglia AL. Solving the bus rapid transit route design problem with general topologies via simultaneous column and cut generation. Technical report COPA-2012-10, Universidad de los Andes; 2012.
[18] Graves GW, McBride RD, Gershkoff I, Anderson D, Mahidhara D. Flight crew scheduling. Management Science 1993;39(6):657–82.
[19] Gromicho J, van Hoorn JJ, Kok AL, Schutten JMJ. Restricted dynamic programming: a flexible framework for solving realistic VRPs. Computers & Operations Research 2011. http://dx.doi.org/10.1016/j.cor.2011.07.002.
[20] Grönkvist M. Accelerating column generation for aircraft scheduling using constraint propagation. Computers & Operations Research 2006;33:2918–34.
[21] Handler GY, Zang I. A dual algorithm for the constrained shortest path problem. Networks 1980;10:293–310.
[22] Joksch H. The shortest route problem with constraints. Journal of Mathematical Analysis and Applications 1966;14(1):191–7.
[23] Lavoie S, Minoux M, Odier E. A new approach for crew pairing problems by column generation with an application to air transportation. European Journal of Operational Research 1988;35:45–58.
[24] Lawler EL. Combinatorial optimization: networks and matroids. New York: Dover Publications; 2001.
[25] Muter İ, Birbil Şİ, Bülbül K, Şahin G, Yenigün H, Taş D, et al. Solving a robust airline crew pairing problem with column generation. Computers & Operations Research 2010. http://dx.doi.org/10.1016/j.cor.2010.11.005.
[26] Giacomo Nannicini, Daniel Delling, Dominik Schultes, Leo Liberti. Bidirectional A^* search on time-dependent road networks. Networks 2012;59(2):240–51.
[27] Reinhardt LB, Pisinger D. Multi-objective and multi-constrained non-additive shortest path problems. Computers & Operations Research 2011;38:605–16.
[28] Restrepo MI, Lozano L, Medaglia AL. Constrained network-based column generation for the multi-activity shift scheduling problem. International Journal of Production Economics. http://dx.doi.org/10.1016/j.ijpe.2012.06.030; 2012.
[29] Santos L, Coutinho-Rodrigues J, Current JR. An improved solution algorithm for the constrained shortest path problem. Transportation Research Part B 2007;41:756–71.
[30] Smith OJ, Boland N, Waterer H. Solving shortest path problems with a weight constraint and replenishment arcs. Computers & Operations Research 2012;39(5):964–84.
[31] Steinzen I, Gintner V, Suhl L, Kliewer N. A time-space network approach for the integrated vehicle- and crew-scheduling problem with multiple depots. Transportation Science 2010;44(3):367–82.
[32] Stojković M, Soumis F, Desrosiers J. The operational airline crew scheduling problem. Transportation Science 1998;32(3):232–45.
[33] Villeneuve D, Desaulniers G. The shortest path problem with forbidden paths. European Journal of Operational Research 2005;165:97–107.
[34] Zhu X, Wilhelm WE. Three-stage approaches for optimizing some variations of the resource constrained shortest-path sub-problem in a column generation context. European Journal of Operational Research 2007;183:564–77.
[35] Zhu X, Wilhelm WE. A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation. Computers & Operations Research 2012;39:164–78.