Optim Lett (2014) 8:2155–2172
DOI 10.1007/s11590-014-0742-x

ORIGINAL PAPER

# Acceleration strategies for the weight constrained shortest path problem with replenishment

**Manuel A. Bolívar · Leonardo Lozano · Andrés L. Medaglia**

Received: 17 July 2013 / Accepted: 28 March 2014 / Published online: 17 April 2014


**Abstract** The weight constrained shortest path problem with replenishment (WCSPP-R) generalizes the constrained shortest path problem (CSP) and has multiple applications in transportation, scheduling, and telecommunications. We present an exact algorithm based on a recursive depth-first search that combines and extends ideas proposed in state-of-the-art algorithms for the CSP and the WCSPP-R. The novelty lies in a set of acceleration strategies that significantly improves the algorithm's performance. We conducted experiments over large real-road networks with up to 6 million nodes and 15 million arcs, achieving speedups of up to 219 times against the state-of-the-art algorithm.

**Keywords** Constrained shortest path problem · replenishment · large-scale networks

## 1 Introduction

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be a directed graph defined by a set of nodes $\mathcal{N} = \{v_1, \ldots, v_i, \ldots, v_n\}$ 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 cost $c_{ij}$, a resource consumption $w_{ij}$, and a binary indicator $r_{ij}$ that takes the value of 1 if the arc is a replenishment arc and takes the value of 0, otherwise. The weight constrained shortest path problem with replenishment (WCSPP-R) consists of

M. A. Bolívar · L. Lozano · A. L. Medaglia (✉)
Universidad de Los Andes, Cr 1E No. 19A-10, ML711, Bogotá, Colombia
e-mail: amedagli@uniandes.edu.co
URL: http://wwwprof.uniandes.edu.co/~amedagli

M. A. Bolívar
e-mail: ma.bolivar643@uniandes.edu.co

L. Lozano
e-mail: leo-loza@uniandes.edu.co

finding the minimum cost path between a start node $v_s \in \mathcal{N}$ and an end node $v_e \in \mathcal{N}$ without exceeding a resource constraint $W$. The WCSPP-R considers replenishment arcs that reset the value of the consumed resource to zero at the tail node, that is, just before traversing the replenishment arc. A feasible path $\mathcal{P}$ in the WCSPP-R is an ordered sequence of nodes that satisfies the resource constraint $W$ everywhere.

The WCSPP-R is a generalization of the well-studied constrained shortest path problem (CSP). For the CSP, Joksch [6] proposed a dynamic programming algorithm that was lately extended with preprocessing techniques by Dumitrescu and Boland [4]. Handler and Zang [5] used a $k$-shortest path algorithm where they identify $k$ paths, sort them by length, and evaluate them successively until finding a feasible path. Santos et al. [10] extended this idea by improving the search direction based on the relative tightness of the resource constraint. More recently, Lozano and Medaglia [8] proposed a recursive depth-first search combined with pruning strategies to avoid a complete exploration of the solution space of the CSP.

In contrast to the CSP, the WCSPP-R has not been studied, with the notable exception of Smith et al. [11]. Motivated by the existence of several replenishment opportunities in contexts like airline crew pairing, aircraft routing, and crew scheduling, they proposed two exact algorithms. The first algorithm uses a meta-network (or high level network) that exploits the inter-replenishment subpath structure of feasible paths. The second method is a label correcting (LC) algorithm combined with an efficient preprocessing technique. After testing the algorithms over a set of randomly generated grid networks and a set of acyclic networks arising from airline planning applications, the LC algorithm clearly outperformed the meta-network based method. This LC algorithm comprises two stages. The first stage is a preprocessing procedure that finds the minimum cost paths $\mathcal{P}^c_{si}$ and $\mathcal{P}^c_{ie}$ from $v_s$ to all nodes $v_i \in \mathcal{N}$ and from every node $v_i \in \mathcal{N}$ to the end node $v_e$; then, it finds the minimum weight feasible paths $\mathcal{P}^w_{si}$ and $\mathcal{P}^w_{ie}$ and aggressively removes from the network those nodes and arcs that cannot be part of the optimal solution. The second stage is a labeling algorithm where a label at node $v_i$ represents a partial path $\mathcal{P}$ from $v_s$ to $v_i$; the label stores the node $v_i$, the cumulative cost $c(\mathcal{P})$, and the resource consumed since the last replenishment $w(\mathcal{P})$. The LC algorithm starts with an empty set of untreated labels that is triggered in $v_s$ setting a label with cost and resource consumption equal to zero. Labels are pulled out from the untreated label set according to a given label treatment criterion and extended from their node along each outgoing arc, generating new labels but discarding infeasible or dominated ones. The LC algorithm stops when the untreated label set is empty.

Equivalently, the pulse algorithm for the CSP [8] also comprises two stages: (1) a bounding stage that finds lower bounds on the cost and the resource consumption from any node $v_i$ to the end node $v_e$, and (2) a recursive exploration stage that finds the optimal solution based on an implicit enumeration of the solution space. The exploration is started by sending a *pulse* from the start node $v_s$. The pulse tries to propagate throughout the outgoing arcs of each visited node recursively, storing at each node the partial path $\mathcal{P}$, the cumulative cost $c(\mathcal{P})$, and the cumulative resource consumption $w(\mathcal{P})$. At each node, different pruning strategies try to prevent pulse propagation based on partial path dominance, infeasibility, and bounds. Every pulse that reaches the end node $v_e$ contains all the information of a feasi-

ble path from $v_s$ to $v_e$ and it is a candidate solution to update the global primal bound.

It is worth noting the similarities between these independent research fronts. First, both algorithms find cost and resource bounds for all nodes. Nonetheless, the preprocessing of the LC algorithm goes a step further and uses the computed information to remove nodes and arcs from the graph. Second, the ideas of discarding a label or pruning a pulse are very similar. Finally, both algorithms use analogous strategies to prevent the propagation of labels and pulses, i.e., infeasibility, dominance, and bounds. However, there are a couple of key differences. First, while the LC algorithm explores all the successors of a label's node and then globally selects the next label to be extended following a lexicographic breadth-first search [1], the pulse algorithm follows a pure depth-first search until the pulse reaches the end node or until it is pruned. Second, in contrast to LC, the pulse algorithm does not require an exhaustive dominance check for correctness, but it heavily relies on the strength of the pruning strategies.

The contribution of this paper is twofold: from a methodological perspective, we present a set of acceleration strategies that combine depth and breadth search, generalizing the ideas proposed by Smith et al. [11] and Lozano and Medaglia [8], both state-of-the-art algorithms for the WCSPP-R and CSP, respectively. From a computational perspective, we adapted a vast set of real-road networks for the WCSPP-R and conducted extensive computational experiments that showed remarkable speedups of up to 219 times against the state-of-the-art algorithm. Moreover, we incorporated the proposed acceleration strategies into the pulse algorithm for the CSP and achieved speedups of up to 23 % on instances from the literature.

The remainder of this paper is organized as follows. Section 2 presents an overview of the pulse algorithm and outlines the intuition behind the acceleration strategies. Section 3 provides a detailed description of the acceleration strategies. Section 4 presents the computational results for the WCSPP-R. Section 5 presents a sensitivity analysis on the individual effect of the acceleration strategies. Section 6 presents additional computational results on the CSP. Finally, Sect. 7 concludes the paper and outlines future work.

## 2 An overview of the proposed algorithm

We extend the pulse algorithm presented by Lozano and Medaglia [8] for the CSP integrating key ideas by Smith et al. [11] for the WCSPP-R and adding three new acceleration strategies, namely, *path completion*, *pulse queueing*, and *best-promise exploration order*. Our approach brings together the best from both algorithms under the pulse framework. In a first stage, our algorithm obtains primal and dual bounds using the flawless preprocessing procedure by Smith et al. [11]. In a second stage, our algorithm explores the network using a modified pulse algorithm that includes the acceleration strategies. A major modification to the original pulse algorithm is the inclusion of a pulse queue denoted by $\mathcal{Q}$. When the depth of a partial path $\mathcal{P}$ reaches a maximum allowed value $\delta$, its exploration pauses and the corresponding pulse is stored in $\mathcal{Q}$, saving the partial path $\mathcal{P}$, the node where the pulse was paused $n(\mathcal{P})$,

the cumulative cost $c(\mathcal{P})$, and the resource consumption $w(\mathcal{P})$. The algorithm stops when the queue $\mathcal{Q}$ is empty. Note that how the algorithm explores the graph depends on the value of $\delta$: if $\delta$ is equal to one, the exploration becomes a pure lexicographic breadth-first search; if $\delta$ is large enough, the exploration becomes a pure depth-first search; and for $1 < \delta < \infty$, the algorithm combines depth-first and breadth-first search strategies.

Algorithm 1 presents the pseudocode of the proposed algorithm. Lines 1 and 2 initialize the optimal path $\mathcal{P}^*$ and a partial path $\mathcal{P}$. Line 3 executes the preprocessing algorithm proposed by Smith et al. [11]. Line 4 initializes the pulse queue $\mathcal{Q}$ with a paused pulse at node $v_s$. Lines 5 through 10 propagate pulses stored in $\mathcal{Q}$. Line 6 selects the next pulse to be processed given a queue discipline (see Sect. 3.4) and removes it from $\mathcal{Q}$. Line 7 checks if the pulse can be discarded by bounds pruning. If the pulse is not discarded, line 8 propagates the pulse resuming the exploration at node $n(\mathcal{P})$ and setting the path depth to zero. Finally, line 11 returns the optimal path $\mathcal{P}^*$ which is obtained (and modified) through the recursion.

**Algorithm 1** Pulse algorithm

**Input:** $\mathcal{G}$, directed graph; $W$, resource constraint; $v_s$, start node; $v_e$, end node.
**Output:** $\mathcal{P}^*$, optimal path.

```
1: P* ← ∅
2: P ← {v_s}
3: preprocess(G, W, v_s, v_e)
4: push(Q, P)
5: while Q ≠ ∅ do
6:     P ← pop(Q)                                   ▷ see Section 3.4
7:     if checkBounds(n(P), c(P)) = false then      ▷ see Section 3.1
8:         pulse(n(P), c(P), w(P), 0, P)            ▷ see Algorithm 2
9:     end if
10: end while
11: return P*
```

Algorithm 2 shows the body of the recursive function pulse, where $\Gamma^+(v_i) = \{v_j \in \mathcal{N} | (i, j) \in \mathcal{A}\}$ is the set of head nodes of the outgoing arcs of node $v_i$. Every time the pulse function is called over the end node $v_e$, the pulse propagation stops, the primal bound is updated, and the information of the best path known is stored globally. Lines 2 through 5 update the cumulative cost, the resource consumption, pulse depth, and partial path. Line 6 checks if it is feasible to reach the end node $v_e$ from node $v_j$ given a resource consumption of $w'$. 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 the list of non-dominated partial paths $\mathcal{L}(v_j)$ for node $v_j$. If $\mathcal{P}'$ is not dominated, it is inserted into $\mathcal{L}(v_j)$ following the same queue discipline of $\mathcal{Q}$ and any path dominated by $\mathcal{P}'$ is removed from $\mathcal{L}(v_j)$. Line 9 explores the possibility of completing the partial path $\mathcal{P}'$ with the minimum cost feasible path (see Sect. 3.2). Line 10 checks if the path has reached the maximum depth $\delta$, if so, line 11 adds it to the pulse queue $\mathcal{Q}$. Finally, line 13 recursively propagates the pulse through node $v_j$.

**Algorithm 2** Pulse function

**Input:** $v_i$, current node; $c$, cumulative cost; $w$, cumulative resource; $d$, current depth; $\mathcal{P}$, partial path.
**Output:** `void`

1: **for** $v_j \in \Gamma^+(v_i)$ **do**
2: $c' \leftarrow c + c_{ij}$
3: $w' \leftarrow w \cdot (1 - r_{ij}) + w_{ij}$
4: $d' \leftarrow d + 1$
5: $\mathcal{P}' \leftarrow \mathcal{P} \cup \{v_j\}$
6: **if** `checkFeasibility`$(v_j, w') =$ `true` **then** ▷ *see Section 3.1*
7: **if** `checkBounds`$(v_j, c') =$ `false` **then** ▷ *see Section 3.1*
8: **if** `checkDominance`$(v_j, \mathcal{P}') =$ `false` **then** ▷ *see Section 3.1*
9: **if** `checkCompletePath`$(v_j, c', w') =$ `false` **then** ▷ *see Section 3.2*
10: **if** $d' > \delta$ **then** ▷ *see Section 3.3*
11: `push`$(\mathcal{Q}, \mathcal{P}')$
12: **else**
13: `pulse`$(v_j, c', w', d', \mathcal{P}')$
14: **end if**
15: **end if**
16: **end if**
17: **end if**
18: **end if**
19: **end for**

## 3 Acceleration strategies

In this section we first review the core pruning strategies of the pulse algorithm [8], namely, *infeasibility*, *bounds*, and *dominance pruning*. Then, we present a detailed description of the newly proposed acceleration strategies.

### 3.1 Core pruning strategies

The core pruning strategies are used in both the LC algorithm and the pulse algorithm. The *infeasibility pruning* strategy discards a partial path $\mathcal{P}_{si}$ when it is not possible to reach the end node without exceeding the resource constraint, i.e., $w(\mathcal{P}_{si}) + w(\mathcal{P}^w_{ie}) > W$. The *bounds pruning* strategy uses a primal bound $\bar{c}$ that it is updated with the value of the best solution found so far. If $c(\mathcal{P}_{si}) + c(\mathcal{P}^c_{ie}) \geq \bar{c}$, then path $\mathcal{P}_{si}$ can be safely pruned because a better (or equal) solution has already been found in the exploration. Finally, for *dominance pruning*, let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two partial paths at a given node $v_i \in \mathcal{N}$. It is said that $\mathcal{P}_1$ dominates $\mathcal{P}_2$ if $c(\mathcal{P}_1) \leq c(\mathcal{P}_2)$ and $w(\mathcal{P}_1) < w(\mathcal{P}_2)$; or $c(\mathcal{P}_1) < c(\mathcal{P}_2)$ and $w(\mathcal{P}_1) \leq w(\mathcal{P}_2)$. For further information about these strategies, the reader is referred to Lozano and Medaglia [8] and Smith et al. [11].

### 3.2 Path completion

Given a partial path $\mathcal{P}_{si}$ arriving to node $v_i \in \mathcal{N}$, the path completion strategy adds the minimum cost path $\mathcal{P}^c_{ie}$ from $v_i$ to $v_e$ to the partial path $\mathcal{P}_{si}$, i.e., $\mathcal{P}_{se} = \mathcal{P}_{si} \cup \mathcal{P}^c_{ie}$ and checks if the completed path $\mathcal{P}_{se}$ is feasible and if $c(\mathcal{P}_{se})$ is less than the primal

bound $\bar{c}$. Note that if a path completion occurs, there is no need to explore additional paths beginning with $\mathcal{P}_{si}$ because $\mathcal{P}^c_{ie}$ is already the minimum cost path from $v_i$ to $v_e$ and thus $\mathcal{P}_{se}$ will be the minimum cost path beginning with partial path $\mathcal{P}_{si}$. In this case the pulse associated with $\mathcal{P}_{si}$ can be pruned and the primal bound can be updated. This idea was implemented in the pulse framework following the intuition shared by Smith et al. [11] as a future extension to their algorithm.

Furthermore, if a path cannot be completed using the minimum cost path, there is still a chance to update the primal bound. Given a partial path $\mathcal{P}_{si}$, the path can be completed by adding the minimum-weight feasible path $\mathcal{P}^w_{ie}$ from $v_i$ to $v_e$ to the partial path, i.e., $\mathcal{P}_{se} = \mathcal{P}_{si} \cup \mathcal{P}^w_{ie}$. If $\mathcal{P}_{se}$ is feasible and $c(\mathcal{P}_{se}) < \bar{c}$, the primal bound can be updated, but the pulse associated with path $\mathcal{P}_{si}$ cannot be pruned.

### 3.3 Pulse queueing

Breadth-first and depth-first search have both advantages and limitations [7]. Breadth-first search (BFS) expands the start node through the outgoing arcs, and then processes the nodes by depth level, layer by layer, until it reaches the end node. The main drawbacks of BFS are: (1) the memory requirements to store all the states; and (2) the possible lack of feasible solutions at intermediate steps of the search (i.e., it may take until the last iteration to reach the end node). On the other hand, depth-first search (DFS) processes an outgoing arc from the most recently expanded node until it reaches the end node; then it backtracks to process unexplored outgoing arcs. In contrast to BFS, the only information stored in DFS relates to the currently explored path; thus it requires less memory than BFS. Because DFS favors depth over breadth, it reaches the end node often, thus it generates complete solutions that are used to update the primal bound. However, a major disadvantage of DFS is that it could waste time exploring unpromising regions of the search space before backtracking and correcting poor decisions made at earlier stages of the exploration.

Because it is not obvious on which instances BFS or DFS performs better, we combine both search paradigms in a simple, yet effective way. The idea is to perform a depth-first search from the start node but restricted to a maximum depth $\delta$. When a pulse reaches that depth, the partial path is stored in the pulse queue $\mathcal{Q}$ following a given queue discipline. Once there are no active pulses, the queued pulses are processed until $\mathcal{Q}$ is empty. If the value for $\delta$ is large enough, the exploration behaves as a pure DFS. Thus, if the first extended arc in a path $\mathcal{P}$ is not part of the optimal solution, it may be necessary to explore all the possible paths that include that arc to conclude that extending it was a naïve decision. However, by fixing a low value for $\delta$, the algorithm is able to look ahead and behaves more like BFS, acquiring the capability to timely reroute the exploration after reaching the depth limit.

### 3.4 Best-promise exploration order

The graph exploration order is critical for the algorithm's performance and it is defined by the queue discipline for $\mathcal{Q}$. Given a partial path $\mathcal{P}_{si}$ arriving to node $v_i$, we considered the following queueing disciplines: (1) minimum cost, (2) maximum cost, (3)

minimum weight, (4) maximum weight, and (5) best promise. The first four disciplines were presented by Smith et al. [11] and the latter is our proposed exploration order. We define the promise of a path $\psi(\mathcal{P}_{si})$ as the cumulative cost of a path $\mathcal{P}_{si}$ plus the cost of the minimum cost path from $v_i$ to $v_e$, formally $\psi(\mathcal{P}_{si}) = c(\mathcal{P}_{si}) + c(\mathcal{P}^c_{ie})$. Given that $\mathcal{P}^c_{ie}$ is the best possible path from node $v_i$ to $v_e$, the rationale behind this queue discipline is to explore first those partial paths which promise the best possible objective function, that is, those paths with the minimum $\psi(\mathcal{P}_{si})$.

Finally, it is important to note that each one of the acceleration strategies has an important effect on the algorithm's performance; however, its efficiency is not the sum of the individual effects of each strategy, but the interaction among all the acceleration strategies and the core pruning strategies.

## 4 Computational experiments

We conducted a head-to-head comparison between our pulse algorithm and the LC algorithm by Smith et al. [11]. Both algorithms were coded in Java, using Eclipse SDK version 3.6.1 and tested on a computer with an Intel Xeon X5450 @ 3.00 GHz (four cores) with 6 GB of RAM allocated to the memory heap size of the Java Virtual Machine on Windows Vista Professional.

We designed two sets of computational experiments with different goals. The first set validates our implementation of the LC algorithm, showing that it is efficient and suitable as a benchmark for comparison. The second set assesses the contributions of the acceleration strategies over a vast set of real-road networks.

### 4.1 The testbed

For the validation experiment we use randomly generated grid networks following the procedure outlined by Smith et al. [11]. We generated 30 instances for grids of $100 \times 100$, $200 \times 200$, and $400 \times 400$ nodes for a grand total of 90 instances. Each instance has a start node with outgoing arcs to the first layer of the grid and an end node with incoming arcs from the last layer.

For the second experiment we used real road networks, three from Raith and Ehrgott [9] and 10 from the 9th DIMACS implementation challenge for the shortest path problem [2]. These networks were divided in two sets based on their size: medium and large. The medium-sized networks range from 9,500 to 436,000 nodes while the large-sized networks contain instances with over 1,000,000 nodes and up to 6,262,104 nodes and 15 million arcs. Following the same approach by Smith et al. [11], each network was adapted to the WCSPP-R by randomly selecting replenishment arcs with a probability of 5 %. For each adapted network, we generated 30 instances with randomly selected start and end nodes for a grand total of 390 instances.

We tested different levels of tightness for the resource constraint. Smith et al. [11] define the resource limit as $W = \alpha W^- + (1-\alpha) W^+$ where $W^-$ is the smallest amount of resource for which there is a feasible path, $W^+$ is the smallest amount of resource for which the minimum cost path is the optimal solution, and $\alpha \in [0, 1]$. We used four values for the tightness factor $\alpha$: 0.1, 0.5, 0.9, and 1; where small (large) values for $\alpha$ lead

**Table 1** Validation of our implementation against the original LC

| Network | $\alpha$ | LC (s) | LC/Java (s) |
|---|---|---|---|
| $100 \times 100$ | 0.1 | 0.21 | 0.02 |
| | 0.5 | 0.19 | 0.04 |
| | 0.9 | 0.13 | 0.10 |
| | 1 | 0.12 | 0.01 |
| $200 \times 200$ | 0.1 | 7.96 | 0.09 |
| | 0.5 | 7.85 | 0.37 |
| | 0.9 | 5.25 | 0.58 |
| | 1 | 4.67 | 0.07 |
| $400 \times 400$ | 0.1 | 10.41 | 0.39 |
| | 0.5 | 9.56 | 3.08 |
| | 0.9 | 8.62 | 3.85 |
| | 1 | 8.05 | 0.30 |

to loosely (tightly) constrained problems. Additionally, we characterized each instance by computing the exact values of $W^-$ and $W^+$. We have made publicly available all instances used in this paper for the WCSPP-R at http://hdl.handle.net/1992/1159.

### 4.2 Validation experiment

Smith et al. [11] tested the performance of 10 different label treatment criteria and concluded that the minimum $c(\mathcal{P})$ is a strong alternative to solve the WCSPP-R. For this reason, we also chose this criterion in our Java implementation of the LC algorithm (henceforth called LC/Java).

Table 1 compares the time (including preprocessing) reported by Smith et al. [11] with their implementation of the LC algorithm against our LC/Java implementation over the set of randomly generated grids. For the sake of fairness, we scaled all our times using the LINPACK benchmark [3]; according to this benchmark our computer is 1.54 times faster than the one used by Smith et al. [11]. Column 1 shows the name of the instance (size of the grid), column 2 presents the tightness of the resource constraint $\alpha$, column 3 shows the average time in seconds reported by Smith et al. [11], and column 4 reports the scaled average time in seconds for LC/Java over the 30 instances generated for each grid configuration.

Table 1 shows that LC/Java is a strong and robust implementation of the algorithm. For all instances and values of $\alpha$, LC/Java outperforms LC. Moreover, when the network size increases, the time gaps become even larger. This good performance is due to a thorough selection of the data structures and a careful implementation of the procedures used for managing the set of labels.

### 4.3 Pulse vs. LC/Java

After having checked that our Java implementation of the LC algorithm by Smith et al. [11] is a valid contender, we conducted a head-to-head comparison between

**Table 2** LC/Java vs. pulse on medium-size real road networks

| Network | Nodes | Arcs | $\alpha$ | Time$_{pre}(s)$ | LC/Java (s) | Pulse (s) | Avg. speedup |
|---|---|---|---|---|---|---|---|
| DC | 9,559 | 39,377 | 0.1 | 0.01 | <0.01 | <0.01 | 1.01 |
| Washington DC | | | 0.5 | 0.01 | <0.01 | <0.01 | 2.58 |
| | | | 0.9 | 0.01 | <0.01 | <0.01 | 2.19 |
| | | | 1 | 0.01 | <0.01 | <0.01 | 1.02 |
| RI | 53,658 | 192,084 | 0.1 | 0.12 | <0.01 | <0.01 | 1.14 |
| Rhode Island | | | 0.5 | 0.12 | 0.02 | 0.01 | 4.54 |
| | | | 0.9 | 0.12 | 0.05 | 0.03 | 7.30 |
| | | | 1 | 0.11 | 0.05 | 0.04 | 2.35 |
| NY | 264,346 | 733,846 | 0.1 | 0.47 | <0.01 | <0.01 | 5.90 |
| New York | | | 0.5 | 0.47 | 0.05 | <0.01 | 15.10 |
| | | | 0.9 | 0.45 | 0.11 | 0.08 | 2.98 |
| | | | 1 | 0.43 | 0.11 | 0.10 | 1.46 |
| BAY | 321,270 | 800,172 | 0.1 | 0.54 | 0.02 | <0.01 | 15.18 |
| San Francisco | | | 0.5 | 0.54 | 0.10 | 0.01 | 41.23 |
| Bay Area | | | 0.9 | 0.50 | 0.11 | 0.09 | 3.66 |
| | | | 1 | 0.47 | 0.15 | 0.17 | 1.40 |
| NJ | 330,386 | 1, 202,458 | 0.1 | 1.29 | <0.01 | <0.01 | 1.03 |
| New Jersey | | | 0.5 | 1.16 | 0.20 | 0.03 | 58.20 |
| | | | 0.9 | 0.86 | 0.75 | 0.42 | 7.47 |
| | | | 1 | 0.76 | 0.50 | 0.34 | 2.27 |
| COL | 435,666 | 1,057,066 | 0.1 | 0.75 | 0.06 | <0.01 | 23.19 |
| Colorado | | | 0.5 | 0.75 | 0.15 | 0.02 | 34.36 |
| | | | 0.9 | 0.76 | 0.24 | 0.13 | 7.16 |
| | | | 1 | 0.84 | 0.36 | 0.31 | 3.42 |
| Overall avg. | | | | | | | 10.26 |

LC/Java and our pulse algorithm. Given that the preprocessing procedure is the same for both algorithms, we decided to separate the preprocessing time from the rest of the execution time. For these experiments, LC/Java still uses the minimum $c(\mathcal{P})$ as label treatment criterion, while the pulse follows an exploration order by best promise $\psi(\mathcal{P})$. After fine tuning the pulse algorithm, the depth limit $\delta$ was set to 2.

Tables 2 and 3 summarize the results of this head-to-head comparison. For each network-tightness configuration, we solved 30 instances with randomly selected start and end nodes for a grand total of 1,560 ($=13 \times 4 \times 30$) experiments. We calculated the computational time with a precision of 1/100 s, meaning that any time under 0.01 is reported as <0.01. Columns 1 through 3 show the name, nodes, and arcs of the network; column 4 presents the tightness of the resource constraint $\alpha$; column 5 presents the average time in seconds spent in the preprocessing procedure (same for LC/Java and for pulse); columns 6 and 7 present the average time in seconds used by LC/Java and pulse after the preprocessing procedure; and column 8 presents the arithmetic mean of the individual speedups calculated as the ratio between the time for LC/Java and the

**Table 3** LC/Java vs. pulse on large-size real road networks

| Network | Nodes | Arcs | $\alpha$ | Time$_{pre}$ (s) | LC/Java (s) | Pulse (s) | Avg. speedup |
|---|---|---|---|---|---|---|---|
| FLA | 1,070,376 | 2,712,798 | 0.1 | 2.28 | 0.23 | $<0.01$ | 36.22 |
| Florida | | | 0.5 | 2.34 | 0.60 | 0.06 | 42.27 |
| | | | 0.9 | 2.30 | 1.05 | 0.73 | 3.43 |
| | | | 1 | 2.28 | 0.86 | 0.89 | 1.55 |
| NW | 1,207,945 | 2,840,208 | 0.1 | 2.46 | 0.14 | $<0.01$ | 38.42 |
| Northwest USA | | | 0.5 | 2.48 | 0.45 | 0.05 | 46.19 |
| | | | 0.9 | 2.41 | 0.65 | 0.29 | 5.79 |
| | | | 1 | 2.31 | 0.86 | 0.89 | 1.79 |
| NE | 1,524,453 | 3,897,636 | 0.1 | 3.62 | 0.12 | $<0.01$ | 33.21 |
| Northeast USA | | | 0.5 | 3.50 | 0.87 | 0.04 | 57.80 |
| | | | 0.9 | 3.54 | 1.09 | 0.91 | 4.41 |
| | | | 1 | 3.24 | 1.22 | 1.13 | 1.33 |
| CAL | 1,890,815 | 4,657,742 | 0.1 | 4.60 | 0.07 | $<0.01$ | 53.00 |
| California and | | | 0.5 | 4.80 | 1.38 | 0.04 | 127.59 |
| Nevada | | | 0.9 | 5.17 | 2.04 | 1.10 | 6.95 |
| | | | 1 | 5.44 | 3.76 | 1.39 | 2.84 |
| LKS | 2,758,119 | 6,885,658 | 0.1 | 6.71 | 0.47 | 0.05 | 27.83 |
| Great Lakes | | | 0.5 | 7.86 | 4.02 | 0.31 | 98.57 |
| | | | 0.9 | 11.04 | 3.35 | 3.26 | 4.38 |
| | | | 1 | 8.43 | 2.95 | 3.70 | 1.55 |
| E | 3,598,623 | 8,778,114 | 0.1 | 14.83 | 0.29 | $<0.01$ | 219.64 |
| Eastern USA | | | 0.5 | 13.35 | 2.32 | 0.16 | 55.12 |
| | | | 0.9 | 16.03 | 3.63 | 2.90 | 4.24 |
| | | | 1 | 12.73 | 2.69 | 4.42 | 1.26 |
| W | 6,262,104 | 15,248,146 | 0.1 | 43.38 | 0.82 | 0.04 | 145.46 |
| Western USA | | | 0.5 | 62.96 | 5.04 | 0.56 | 100.81 |
| | | | 0.9 | 64.25 | 8.59 | 2.38 | 6.69 |
| | | | 1 | 57.68 | 11.99 | 11.66 | 1.59 |
| Overall avg. | | | | | | | 40.36 |

pulse for each instance (note that this value is not the ratio between columns 4 and 5, but the average among the 30 experiments for each network-tightness configuration). Finally, the last row presents an overall average of the speedups over the whole testbed.

The results presented in Table 2 show that the pulse consistently outperforms LC/Java in 23 out of 24 midsized real-road network instances and through all values of the tightness factor $\alpha$. The acceleration strategies lead to average speedups of up to 58 and an average speedup of roughly 10, proving that these strategies really pay back on real-road networks.

Table 3 shows that the pulse also outperforms LC/Java in all the experiments made on large sized instances and across all values of the tightness factor $\alpha$. Note that the average speedups in these networks are larger than those obtained in the midsized

networks, thus showing a good sign of scalability of the pulse algorithm. In fact, the average speedups increase up to 219 times; and the overall speedup by the pulse is roughly 40.

## 5 Sensitivity analysis

We conducted additional experiments to measure the effect of the proposed acceleration strategies on the performance of the pulse algorithm. For this purpose, we picked a medium-size (New Jersey) and a large-size (California and Nevada) road network with their 30 random instances (with different start and end nodes) and four tightness factors for $\alpha$, namely, 0.1, 0.5, 0.9, and 1 (where small values represent loosely constrained instances).

### 5.1 *Marginal analysis*

We measured the marginal effect of each acceleration strategy on the algorithm's performance. Let $\Delta_i := \frac{t_i - t_p}{t_i} \times 100\,\%$ be the time reduction (in percentage) due to the acceleration strategy $i$, where $t_i$ is the runtime of the pulse algorithm without the acceleration strategy $i$, and $t_p$ is the runtime including all the acceleration strategies. The marginal contribution of the path completion strategy was isolated by completely removing the strategy from the algorithm and comparing against the benchmark pulse which includes it. Similarly, the marginal effect of the pulse queue was isolated by setting a depth limit $\delta$ to a very large value ($\delta = 100$) and comparing against the default value ($\delta = 2$). On the other hand, the effect of the best-promise exploration order is inherently tied to the use of the pulse queue, so we defer this discussion until Sect. 5.2.2.

Table 4 summarizes the results of this marginal-analysis experiment. Column 1 identifies the network; column 2 presents the tightness factor $\alpha$ for the resource constraint; and columns 3 and 4 show the time reduction $\Delta_i$ (in percentage) for the path completion ($i = \text{PC}$) and pulse queue strategies ($i = \text{PQ}$), respectively. Finally, the last row shows the overall time reduction due to each strategy over the whole testbed comprised of 240 ($=2 \times 4 \times 30$) runs.

Table 4 shows that including the path completion strategy speeds up the pulse algorithm on average by 9.2 %; also, this strategy is responsible for average time reductions of up to 28.6 % (for CAL road network with $\alpha = 0.5$). Note, however, that in the specific case of the tightly-constrained CAL road network (with $\alpha = 1$), the path completion strategy has a negative effect on the algorithm's runtime. In this case of a tight resource constraint over a large-size network, the path completion strategy turns out to be overly expensive, because the chances of succeeding (i.e., completing the path) are low. Regarding the pulse queue strategy, Table 4 shows an overall average time reduction of 65.8 %. Moreover, this strategy seems to consistently reach average time reductions of over 96 % for the tighter instances (with $\alpha = 0.9$ and $\alpha = 1$) for both networks, suggesting that the contribution of the pulse queue is essential for the pulse performance while solving the WCSPP-R. In the next subsections we delve

**Table 4** Average time reduction due to each acceleration strategy

| Network | $\alpha$ | Path completion ($\Delta_{PC}$) (%) | Pulse queue ($\Delta_{PQ}$) (%) |
|---|---|---|---|
| NJ | 0.1 | 2.1 | 2.1 |
| New Jersey | 0.5 | 10.0 | 43.7 |
| | 0.9 | 9.9 | 96.1 |
| | 1 | 6.3 | 97.7 |
| CAL | 0.1 | 18.8 | 13.7 |
| California and Nevada | 0.5 | 28.6 | 76.0 |
| | 0.9 | 2.7 | 98.7 |
| | 1 | −4.9 | 98.6 |
| Overall avg. | | 9.2 | 65.8 |

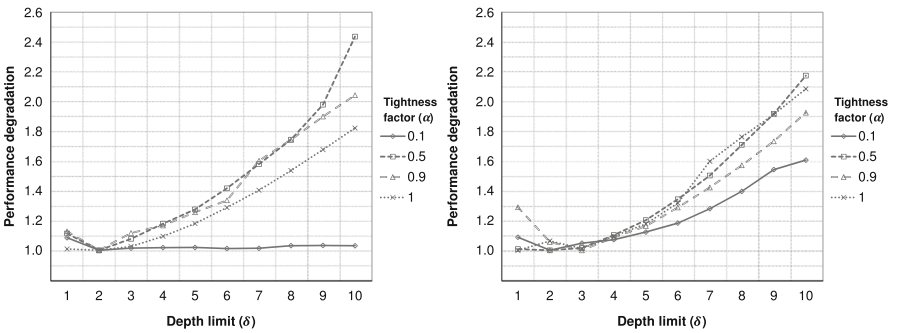
deeper into the two main components of the pulse queue, namely, the *depth limit* and the *queue discipline* (exploration order).

## 5.2 Pulse queue strategy

### *5.2.1 Depth limit*

We conducted a sensitivity analysis to evaluate the impact on the algorithm's performance by changing the depth limit $\delta$ of the pulse queue. We defined a performance degradation metric as the ratio between the average time achieved with a given value of $\delta$ and the minimum average time achieved among all values of $\delta$. For a given value of $\delta$, a performance degradation of 1 means that this value of $\delta$ achieved the minimum average time. Figure 1 shows the results of this experiment over different values of the tightness factor $\alpha$ of the resource constraint.

For the medium-size road network (see Fig. 1a), the algorithm achieved its best performance with a depth limit of $\delta = 2$, independently of the tightness factor $\alpha$. In




**(a)** New Jersey (NJ) road network **(b)** California and Nevada (CAL) road network

**Fig. 1** Sensitivity analysis on the pulse depth limit ($\delta$)

general, as the depth limit $\delta$ grows, the algorithm's performance degrades, except for $\alpha = 0.1$ (i.e., the loose resource scenario). With respect to the large-size road network (see Fig. 1b), when the resource constraint is tight ($\alpha = 0.9$ and $\alpha = 1$) the best value for $\delta$ is 3, while for other levels of tightness ($\alpha = 0.1$ and $\alpha = 0.5$), the best value for $\delta$ turns out to be 2. For $\delta \geq 3$, the performance degradation follows a similar increasing trend over all values of $\alpha$ as in the medium-size network. In conclusion, it follows that for these real road network topologies, small depth limits ($\delta = 2$ or $\delta = 3$) seem to achieve a good performance, regardless of the tightness factor.

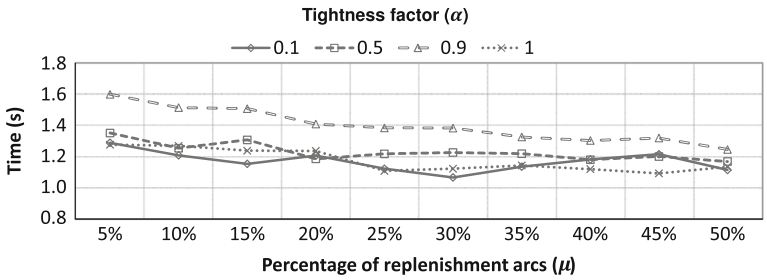
### *5.2.2 Exploration order*

The graph exploration order is critical not only for the proposed pulse algorithm (embedded in its queue discipline), but also for the LC algorithm. We conducted an experiment to measure the effect of using best promise (BP) against minimum cost (MC) exploration order in both the LC and pulse algorithms. Table 5 shows the results for this experiment, where column 1 identifies the network; column 2 shows the tightness factor for the resource constraint; columns 3 and 6 present the average time in seconds using the MC exploration order for LC/Java and pulse, respectively; columns 4 and 7 present the average time in seconds using the BP exploration order for LC/Java and pulse, respectively; and finally, columns 5 and 8 show the average speedups calculated as the average of the ratios between the times using the MC and BP exploration orders (for each instance) for LC/Java and pulse, respectively.

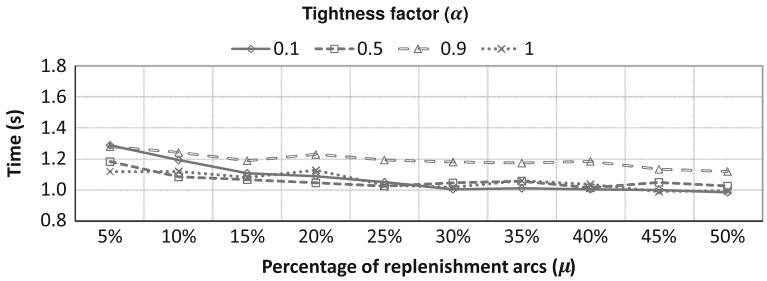
Table 5 shows that BP outperforms MC over all algorithms, instances, and tightness factors. Furthermore, BP leads to average speedups of up to 107 in the LC/Java algorithm and up to 48 in the pulse algorithm, when compared against the MC exploration order. Moreover, BP is on average 24 and 8 times faster than MC in LC/Java and pulse, respectively. Hence, the experiment suggests that the proposed best promise exploration order significantly improves the performance for both algorithms.

**Table 5** Comparison between minimum cost (MC) and best promise (BP) exploration order in LC/Java and pulse algorithms

| Network | $\alpha$ | LC/Java | | | Pulse | | |
|---|---|---|---|---|---|---|---|
| | | MC (s) | BP (s) | Avg. speedup | MC (s) | BP (s) | Avg. speedup |
| NJ | 0.1 | $<0.01$ | $<0.01$ | 1.02 | $<0.01$ | $<0.01$ | 1.06 |
| New Jersey | 0.5 | 0.20 | 0.03 | 49.63 | 0.08 | 0.03 | 10.69 |
| | 0.9 | 0.75 | 0.50 | 6.74 | 0.38 | 0.42 | 1.82 |
| | 1 | 0.50 | 0.37 | 2.11 | 0.30 | 0.34 | 0.99 |
| CAL | 0.1 | 0.07 | $<0.01$ | 18.14 | $<0.01$ | $<0.01$ | 2.26 |
| California and Nevada | 0.5 | 1.38 | 0.05 | 107.28 | 0.53 | 0.04 | 48.37 |
| | 0.9 | 2.04 | 1.31 | 6.34 | 1.16 | 1.10 | 1.84 |
| | 1 | 3.76 | 1.51 | 2.58 | 1.19 | 1.39 | 1.03 |
| Overall avg. | | | | 24.23 | | | 8.51 |




**(a)** LC/Java




**(b)** Pulse

**Fig. 2** LC/Java (**a**) and pulse (**b**) performance over different fractions of replenishment arcs in the New Jersey (NJ) road-network

### 5.3 Replenishment arcs

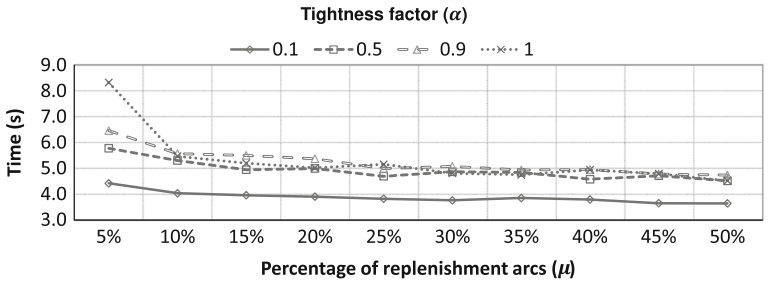
We conducted an experiment to measure the impact on the algorithms' performance by varying the fraction of replenishment arcs. For each network, we randomly selected replenishment arcs with probability $\mu = 0.05, 0.10, ..., 0.50$; and we generated 30 instances with the same start and end node across all values of $\mu$.

Figures 2 and 3 illustrate the average runtimes (including preprocessing time) for the pulse and LC/Java algorithms while increasing the value of $\mu$ from 0.05 to 0.5 with different values for the tightness factor $\alpha$.

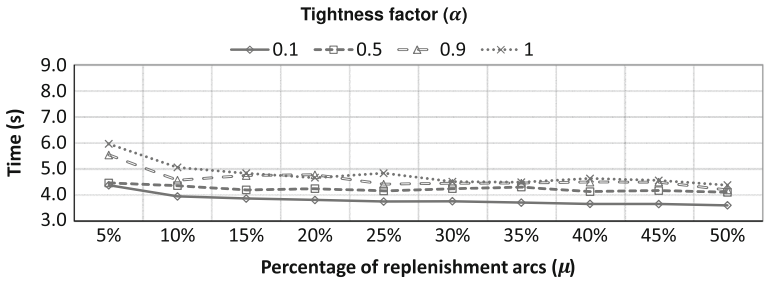
As the fraction of replenishment arcs increases, both algorithms tend to perform better. We believe that this trend can be explained by noting that for higher values of $\mu$ the resource constraint is easier to satisfy due to the large amount of replenishment arcs in the network. Therefore, for larger values of $\mu$ the problem resembles an unconstrained shortest path problem which is easier to solve.

## 6 Accelerating the pulse algorithm for the CSP

Given that the CSP is a particular case of the WCSPP-R, we measured the effect of incorporating the proposed acceleration strategies in the pulse algorithm to solve the CSP.




**(a)** LC/Java




**(b)** Pulse

**Fig. 3** LC/Java (**a**) and pulse (**b**) performance over different fractions of replenishment arcs in the California and Nevada (CAL) road-network

In this special case, the preprocessing procedure does not require the complex features to deal with replenishment arcs and instead it can rely on one-to-all shortest paths to compute the minimum cost path and the minimum weight path between all nodes $v_i \in \mathcal{N}$ and the end node $v_e$ [8]. Likewise, the pulse algorithm was fine-tuned fixing the depth limit to $\delta = 3$, for the CSP.

As for the testbed for the CSP, we used the 180 networks proposed by Santos et al. [10] and also used by Lozano and Medaglia [8]. The testbed is organized in three groups: (1) the (relatively) small networks with $|\mathcal{N}| = 10{,}000$ and $|\mathcal{A}| = 15{,}000$, 25,000, 50,000, 100,000, 150,000 and 200,000; (2) the middle-sized networks with $|\mathcal{N}| = 20{,}000$ and $|\mathcal{A}| = 30{,}000$, 50,000, 100,000, 200,000, 300,000 and 400,000; and (3) the large-sized instances with $|\mathcal{N}| = 40{,}000$ and $|\mathcal{A}| = 60{,}000$, 100,000, 200,000, 400,000, 600,000 and 800,000. For each combination of $|\mathcal{N}| - |\mathcal{A}|$ in each size group, there are 10 different instances. Additionally, we used five values for the constraint tightness factor $\alpha$ : 0.2, 0.4, 0.6, 0.8 and 0.9, where small (large) values mean loosely (tightly) constrained instances.

Table 6 compares the performance of the original pulse algorithm (Pulse-O) against the version that includes the acceleration strategies (Pulse-A). The rows are organized in three categories, namely, small, medium and large-sized networks. Columns 1 and 2 show the number of nodes and arcs in each network. The remaining columns show for different values of $\alpha$, the average runtime for Pulse-O in seconds, the average runtime for Pulse-A in seconds, and the arithmetic mean of the individual speedups calculated

**Table 6** Measuring the effect of the acceleration strategies over the Santos et al. [10] testbed for the CSP

| Nodes | Arcs | $\alpha = 0.2$ | | | $\alpha = 0.4$ | | | $\alpha = 0.6$ | | | $\alpha = 0.8$ | | | $\alpha = 0.9$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Pulse-O | Pulse-A | Speedup | Pulse-O | Pulse-A | Speedup | Pulse-O | Pulse-A | Speedup | Pulse-O | Pulse-A | Speedup | Pulse-O | Pulse-A | Speedup |
| 10,000 | 15,000 | < 0.01 | < 0.01 | 1.03 | < 0.01 | < 0.01 | 1.02 | 0.01 | 0.01 | 1.02 | < 0.01 | < 0.01 | 1.02 | 0.02 | 0.01 | 1.02 |
| 10,000 | 25,000 | 0.01 | 0.01 | 1.02 | 0.01 | 0.01 | 1.02 | 0.01 | 0.01 | 1.02 | 0.01 | 0.01 | 1.01 | 0.01 | 0.01 | 1.01 |
| 10,000 | 50,000 | 0.02 | 0.02 | 1.01 | 0.02 | 0.02 | 1.00 | 0.02 | 0.02 | 1.01 | 0.02 | 0.02 | 1.00 | 0.02 | 0.02 | 1.00 |
| 10,000 | 100,000 | 0.03 | 0.03 | 1.02 | 0.03 | 0.03 | 1.03 | 0.03 | 0.03 | 1.02 | 0.03 | 0.03 | 1.01 | 0.03 | 0.03 | 1.00 |
| 10,000 | 150,000 | 0.05 | 0.05 | 1.05 | 0.06 | 0.05 | 1.12 | 0.05 | 0.05 | 1.10 | 0.05 | 0.05 | 1.06 | 0.05 | 0.05 | 1.02 |
| 10,000 | 200,000 | 0.07 | 0.06 | 1.06 | 0.07 | 0.07 | 1.14 | 0.08 | 0.07 | 1.16 | 0.08 | 0.07 | 1.14 | 0.06 | 0.06 | 1.02 |
| 20,000 | 30,000 | 0.02 | 0.02 | 1.00 | 0.02 | 0.02 | 1.00 | 0.02 | 0.02 | 1.00 | 0.02 | 0.02 | 1.00 | 0.02 | 0.02 | 1.00 |
| 20,000 | 50,000 | 0.03 | 0.03 | 1.00 | 0.03 | 0.03 | 1.00 | 0.03 | 0.03 | 1.00 | 0.03 | 0.03 | 1.00 | 0.03 | 0.03 | 1.00 |
| 20,000 | 100,000 | 0.04 | 0.04 | 1.01 | 0.04 | 0.04 | 1.01 | 0.04 | 0.04 | 1.01 | 0.04 | 0.04 | 1.00 | 0.04 | 0.04 | 1.00 |
| 20,000 | 200,000 | 0.07 | 0.07 | 1.01 | 0.07 | 0.07 | 1.02 | 0.07 | 0.07 | 1.02 | 0.07 | 0.07 | 1.01 | 0.07 | 0.07 | 1.00 |
| 20,000 | 300,000 | 0.12 | 0.11 | 1.10 | 0.11 | 0.11 | 1.08 | 0.12 | 0.11 | 1.10 | 0.11 | 0.11 | 1.06 | 0.10 | 0.10 | 1.01 |
| 20,000 | 400,000 | 0.14 | 0.14 | 1.05 | 0.16 | 0.14 | 1.20 | 0.17 | 0.14 | 1.23 | 0.15 | 0.14 | 1.10 | 0.14 | 0.13 | 1.04 |
| 40,000 | 60,000 | 0.04 | 0.04 | 1.00 | 0.04 | 0.04 | 1.00 | 0.04 | 0.04 | 1.00 | 0.04 | 0.04 | 1.00 | 0.04 | 0.04 | 1.00 |
| 40,000 | 100,000 | 0.06 | 0.06 | 1.00 | 0.06 | 0.06 | 1.00 | 0.06 | 0.06 | 1.00 | 0.06 | 0.06 | 1.00 | 0.06 | 0.06 | 1.00 |
| 40,000 | 200,000 | 0.10 | 0.09 | 1.00 | 0.10 | 0.09 | 1.00 | 0.09 | 0.09 | 1.00 | 0.09 | 0.09 | 1.00 | 0.10 | 0.10 | 1.00 |
| 40,000 | 400,000 | 0.17 | 0.16 | 1.01 | 0.17 | 0.16 | 1.01 | 0.17 | 0.17 | 1.01 | 0.17 | 0.17 | 1.00 | 0.17 | 0.17 | 1.00 |
| 40,000 | 600,000 | 0.25 | 0.24 | 1.06 | 0.25 | 0.24 | 1.06 | 0.25 | 0.24 | 1.04 | 0.24 | 0.23 | 1.04 | 0.23 | 0.23 | 1.01 |
| 40,000 | 800,000 | 0.35 | 0.32 | 1.12 | 0.38 | 0.32 | 1.18 | 0.39 | 0.34 | 1.16 | 0.37 | 0.34 | 1.09 | 0.33 | 0.32 | 1.03 |
| Overall avg. | | | 1.03 | | | | 1.05 | | | 1.05 | | | 1.03 | | | 1.01 |

as the ratio between the runtimes of both versions of the algorithm. Finally, the last row presents an overall average of speedups for each value of $\alpha$.

Table 6 shows that on average, the acceleration strategies slightly improve (yet never degrade) the performance of the (already fast) pulse algorithm on the CSP. The Pulse-A achieves speedups of up to 23 % and the overall average of speedups ranges between 1 and 5 % for different values of $\alpha$. It is noteworthy that the greater speedups occur when the constraint is not too tight (loose), i.e., $\alpha = 0.4$ and $\alpha = 0.6$.

## 7 Concluding remarks

We extended the pulse algorithm by Lozano and Medaglia [8] integrating ideas from Smith et al. [11] and presenting a set of acceleration strategies for the WCSPP-R. First, we provided a *path completion strategy* that avoids unnecessary exploration of suboptimal regions of the solution space (i.e., paths in the network) and strengthens the primal bound (incumbent). Second, we combined the ideas of performing breadth- and depth-first search via a pulse queue that controls the depth of the exploration. Finally, we proposed a best-promise graph exploration order based on an optimistic assessment drawn from lower bounds.

From a computational perspective, we adapted and characterized in terms of the resource consumption a vast set of 390 real-road networks that now comprises a comprehensive publicly available testbed for the WCSPP-R. We conducted several computational experiments that demonstrate the positive effects of the proposed acceleration strategies. In the midsized networks, the pulse algorithm reached average speedups of up to 58 and was on average 10 times faster than the state-of-the-art algorithm. In the larger networks, the effect of the acceleration strategies is even more noticeable: the pulse algorithm achieved overall speedups of roughly 40 times; and reached average speedups of up to 219 times on networks with up to 6 million nodes and 15 million arcs.

We conducted a comprehensive sensitivity analysis that provides an in-depth view of the acceleration strategies and its positive effect on the pulse performance. We also observed how the computational runtime of the pulse algorithm decreases when the fraction of replenishment arcs increases. In addition, we implemented the acceleration strategies in the pulse algorithm for the CSP achieving speedups of up to 23 % compared to the original pulse algorithm.

Finally, our acceleration strategies are easy to understand and to implement and could be applied both on the pulse algorithm or other label-based algorithms like LC. Work currently underway includes extending the algorithm to incorporate replenishment under uncertainty.

## References

1. Corneil, D.: Lexicographic breadth first search: A survey. Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science, vol. 3353, pp. 1–19. Springer, Berlin, Heidelberg (2005)
2. Demetrescu, C., Goldberg, A., Johnson, D.: 9th DIMACS implementation challenge-shortest paths (2006). www.dis.uniroma1.it/challenge9/

3. Dongarra, J.J.: Performance of various computers using standard linear equations software. Technical report CS-89-85, University of Tennessee, USA (2013)
4. Dumitrescu, I., Boland, N.: Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. Networks **42**(3), 135–153 (2003)
5. Handler, G.Y., Zang, I.: A dual algorithm for the constrained shortest path problem. Networks **10**(4), 293–309 (1980)
6. Joksch, H.C.: The shortest route problem with constraints (shortest route problem with constraint, using set of nodes). J. Math. Anal. Appl. **14**(2), 191–197 (1966)
7. Korf, R.E.: Depth-first iterative-deepening: an optimal admissible tree search. Artif. Intell. **27**(1), 97–109 (1985)
8. Lozano, L., Medaglia, A.L.: On an exact method for the constrained shortest path problem. Comput. Oper. Res. **40**(1), 378–384 (2013)
9. Raith, A., Ehrgott, M.: A comparison of solution strategies for biobjective shortest path problems. Comput. Oper. Res. **36**(4), 1299–1331 (2009)
10. Santos, L., Coutinho-Rodrigues, J., Current, J.R.: An improved solution algorithm for the constrained shortest path problem. Transp. Res. Part B: Methodol. **41**(7), 756–771 (2007)
11. Smith, O.J., Boland, N., Waterer, H.: Solving shortest path problems with a weight constraint and replenishment arcs. Comput. Oper. Res. **39**(5), 964–984 (2012)