



Improved filtering for weighted circuit constraints

Pascal Benchimol, Willem-Jan van Hoeve, Jean-Charles Régin, Louis-Martin
Rousseau, Michel Rueher

► To cite this version:

Pascal Benchimol, Willem-Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 2012, 17 (3), pp.205–233. 10.1007/s10601-012-9119-x . hal-01344070

HAL Id: hal-01344070

<https://hal.science/hal-01344070>

Submitted on 11 Jul 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improved Filtering for Weighted Circuit Constraints

Pascal Benchimol · Willem-Jan van Hoeve ·
Jean-Charles Régin · Louis-Martin Rousseau ·
Michel Rueher

the date of receipt and acceptance should be inserted later

Abstract We study the *weighted circuit* constraint in the context of constraint programming. It appears as a substructure in many practical applications, particularly routing problems. We propose a domain filtering algorithm for the weighted circuit constraint that is based on the 1-tree relaxation of Held and Karp. In addition, we study domain filtering based on an additive bounding procedure that combines the 1-tree relaxation with the assignment problem relaxation. Experimental results on Traveling Salesman Problem instances demonstrate that our filtering algorithms can dramatically reduce the problem size. In particular, the search tree size and solving time can be reduced by several orders of magnitude, compared to existing constraint programming approaches. Moreover, for medium-size problem instances, our method is competitive with the state-of-the-art special-purpose TSP solver Concorde.

1 Introduction

Many practical industrial problems, most importantly routing problems, ask to find a circuit in a weighted graph, typically with additional properties. The most famous example is to find a circuit with minimum total weight, visiting all nodes of the graph exactly once, i.e., the well-known Traveling Salesman Problem (TSP) [2, 27]. Additional properties that are often imposed in practice are time windows in which the nodes must be visited, or precedence relations between the nodes.

For the symmetric Traveling Salesman Problem without side constraints, the most effective exact solution methods are based on integer linear programming technology [27]. The most notable example is the special-purpose TSP solver Concorde, that

Pascal Benchimol, Louis-Martin Rousseau
CIRRELT, École Polytechnique de Montréal, Montreal, Canada
E-mail: {pascal.benchimol,louis-martin.rousseau}@polymtl.ca

Willem-Jan van Hoeve
Tepper School of Business, Carnegie Mellon University, Pittsburgh, USA
E-mail: vanhoeve@andrew.cmu.edu

Jean-Charles Régin, Michel Rueher
I3S-CNRS, University of Nice-Sophia Antipolis, Nice, France
E-mail: jean-charles.regin@unice.fr,rueher@polytech.unice.fr

can solve pure symmetric Traveling Salesman Problems on 10,000s or more of nodes in a reasonable time [2]. However, when side constraints are involved, Concorde can no longer be applied, and current exact methods in general cannot scale beyond, say, a few hundred nodes at best.

One competitive approach to solving such (routing) problems with several side constraints is constraint programming (CP), see for example [22, 37]. In a constraint programming model, different aspects of the problem can be expressed by so-called global constraints that capture the combinatorial structures forming the problem [47, 32]. For example, in the case of the Traveling Salesman Problem with Time Windows, one global constraint may capture the fact that we need to have a closed tour, while another global constraint may capture the time window information. Each such constraint has an associated domain filtering algorithm that removes domain values that are provably inconsistent with that constraint, and hence with the problem as a whole. A global view on the problem is established through constraint propagation, in which the updated variable domains are communicated from one constraint to the next, until a fixed point is reached [6].

In constraint programming, specific syntax for expressing unweighted circuit constraints in a graph have been proposed already since the first CP systems were developed [39, 4]. Most current CP systems contain a constraint to model unweighted circuits, although the associated filtering algorithm may be quite different for each system. Weighted circuit constraints are less common in CP systems, as the weights and the circuit are typically handled separately. However, several filtering algorithms have been proposed in the literature that can be applied to the weighted circuit constraint, for example by Caseau and Laburthe [9], Pesant et al. [42], and Focacci et al. [21] (see Section 3 for more detailed information). Yet, no professional CP system currently offers any of these algorithms, and solving problems that contain weighted circuit constraints remains a challenge for constraint programming. One of the main motivations of our work is to expand the reach of constraint programming solvers to complex routing problems by proposing more effective filtering algorithms for the weighted circuit constraint.

The main contributions of this work are the following. First, we introduce a filtering algorithm for the weighted circuit constraint based on the well-known 1-tree relaxation by Held and Karp [28, 29]. Second, we utilize a ‘set variable’ representation of the constraint, allowing the application of different relaxation-based filtering methods on the same variables. In particular, we study an additive bounding procedure by coupling the 1-tree relaxation with the assignment problem relaxation. Third, we analyze the strengths of the different relaxations in terms of their respective filtering power by extensive computational results. Lastly, we experimentally show that our method currently provides the most competitive constraint programming approach for solving Traveling Salesman Problems.

The paper is structured as follows. In Section 2 we provide necessary definitions and basic methodological concepts. In Section 3 we provide an overview of related work. In Section 4 we introduce our representation of the weighted circuit constraint, and discuss the relationship with other existing representations. In Section 5 we present our filtering algorithms based on the 1-tree relaxation. In Section 6 we present the additive bounding procedure that can be applied to strengthen the filtering algorithm. The computational results are presented in Section 7. We present our main conclusions in Section 8.

2 Preliminaries

In this section we present definitions and methodology on which our methods are based. To make the paper as self-contained as possible we recall graph-theoretic definitions as well as a description of the 1-tree relaxation and the assignment problem relaxation.

2.1 Basic definitions

We first recall some basic definitions. Let $G = (V, E, w)$ be an undirected weighted graph with node set V edge set E , and edge weights $w : E \rightarrow \mathbb{Q}_+$. We let $n = |V|$ and $m = |E|$.

A *circuit* in G is a sequence $C = v_0, e_1, v_1, \dots, e_k, v_k$ where $k \geq 0$, such that

- $v_0, v_1, \dots, v_k \in V$,
- $v_k = v_0$,
- $v_0, v_1, \dots, v_{k-1}$ are all distinct,
- $e_1, e_2, \dots, e_k \in E$, and
- $e_i = (v_{i-1}, v_i)$ for $i = 1, \dots, k$.

When k equals $|V|$, C is called a *Hamiltonian circuit*. We define the weight of C as $w(C) = \sum_{i=1}^k w(e_i)$. Given a nonnegative number K , the *weighted Hamiltonian circuit problem* asks to find a Hamiltonian circuit in G with weight at most K . The (*symmetric*) *Traveling Salesman Problem*, or TSP, asks to find a Hamiltonian circuit in G with minimum weight. In constraint programming, the *weighted circuit* constraint corresponds precisely to the weighted Hamiltonian circuit problem, being a feasibility problem instead of an optimization problem (we present a definition of the weighted circuit constraint in Section 4). We note, however, that the optimal solution of the TSP serves as a certificate to the feasibility of the weighted circuit constraint. Furthermore, optimal solutions to relaxations of the TSP are valid lower bounds for the weighted circuit constraint.

For directed graphs edges are ordered pairs, called *arcs*. We assume without loss of generality that an arc between two nodes occurs at most once. For a directed graph $G = (V, A, w)$ on node set V , arc set A and arc weights $w : A \rightarrow \mathbb{Q}_+$, the above definitions are generalized naturally to directed versions by replacing edge e_i by arc a_i . When the weight function w for a directed graph is not symmetric, the corresponding TSP is referred to as an *asymmetric TSP*.

Recall that for a graph $G = (V, E)$, a *spanning tree* is a connected subgraph of G with $|V| - 1$ edges. For simplicity, we also identify a spanning tree with its set of edges.

Lastly, we fix the following notation. Let $G = (V, E, w)$ be an undirected weighted graph. For an edge $e \in E$, we let $G \setminus \{e\}$ denote the graph $(V, E \setminus \{e\}, w)$. For a node $i \in V$, we let $G \setminus \{i\}$ denote the graph $(V \setminus \{i\}, E \setminus \{(i, j) \mid j \in V\}, w)$.

For $S \subset V$, we let $\delta(S)$ denote the set of all edges (i, j) with $i \in S$ and $j \in V \setminus S$. In particular, we let $\delta(i)$ represent all edges adjacent to node $i \in V$.

2.2 Linear Model for the Traveling Salesman Problem

The TSP can be modeled as an integer linear program by introducing a binary variable x_e for all $e \in E$, representing whether edge e is included in the circuit or not [2]:

$$\min \sum_{e \in E} w(e)x_e \quad (1)$$

$$\text{s.t. } \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V \quad (2)$$

$$\sum_{i,j \in S, i < j} x_{(i,j)} \leq |S| - 1 \quad \forall S \subset V, |S| \geq 3 \quad (3)$$

$$x_e \in \{0, 1\} \quad \forall e \in E \quad (4)$$

Relaxations can be obtained by relaxing the degree constraints (2) or the subtour elimination constraints (3) that define the connectivity. Both relaxations will be discussed next.

2.3 1-Tree Relaxation

The 1-tree relaxation, introduced by Held and Karp [28, 29], follows from relaxing the degree constraints (2).

Without loss of generality, we assume that the nodes V are labeled $\{1, 2, \dots, n\}$. A *1-tree* is defined as a spanning tree on the subgraph induced by the set of nodes $V \setminus \{1\}$, together with two distinct edges incident to node 1. Note that the choice of node 1 is arbitrary, depending on the labeling of V . The degree of a node is the set of edges in the 1-tree incident to that node, and we denote it by $\deg(i)$ for $i \in V$. The 1-tree relaxation asks to find a 1-tree with minimum total edge weight. To see that a minimum 1-tree is a relaxation for the TSP, observe that every tour in the graph is a 1-tree, and if a minimum-weight 1-tree is a tour, it is an (optimal) solution to the TSP. Note that a 1-tree is a tour if and only if the degree of all nodes is two.

The 1-tree relaxation can be obtained by moving the degree constraints (2) for all nodes in $V \setminus \{1\}$ into the objective with associated Lagrangean multipliers $\pi_1, \pi_2, \dots, \pi_n$ (where $\pi_1 = 0$), yielding the following model:

$$\min \sum_{e \in E} w(e)x_e + \sum_{i \in V \setminus \{1\}} \pi_i (2 - \sum_{e \in \delta(i)} x_e) \quad (5)$$

$$\text{s.t. } \sum_{e \in \delta(1)} x_e = 2 \quad (6)$$

$$\sum_{e \in E} x_e = |V| \quad (7)$$

$$\sum_{i,j \in S, i < j} x_{(i,j)} \leq |S| - 1 \quad \forall S \subset V \setminus \{1\}, |S| \geq 3 \quad (8)$$

$$x_e \in \{0, 1\} \quad \forall e \in E \quad (9)$$

The Lagrangean multipliers π are also called *node potentials*. In this model, constraints (8) still define the subtour elimination constraints that together with constraints (6)

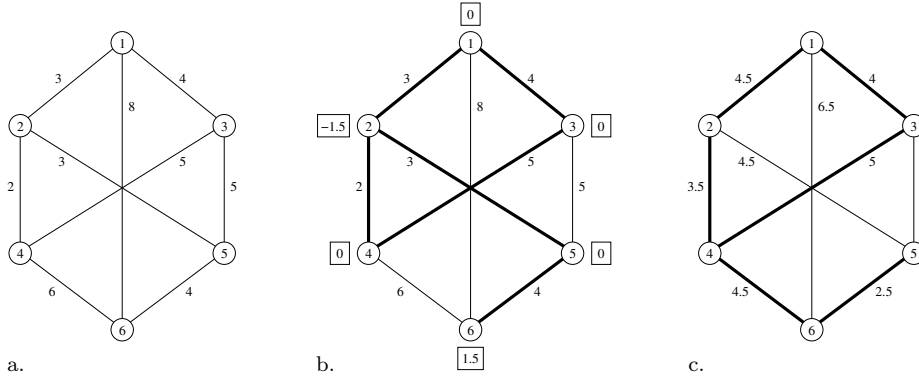


Fig. 1 a. The graph corresponding to Example 1. Edge weights are indicated next to each edge. b. A minimum 1-tree for the edge weights under a., represented by bold edges. Boxed numbers represent the node potentials. The objective value of this 1-tree is 21. c. Updated edge weights and a new minimum 1-tree with objective value 24.

and (7) define the 1-tree structure. An optimal 1-tree relaxation (i.e., with highest objective value) is found by optimizing over the π variables, for example through sub-gradient optimization. Although there exist various alternatives for doing this, in our implementation we closely follow the original combinatorial approach by Held and Karp [28, 29].

The iterative approach proposed by Held and Karp produces a sequence of 1-trees which increasingly resemble tours. We start by computing an initial minimum-weight 1-tree, by finding a minimum-spanning tree on $G \setminus \{1\}$, and adding the two edges with lowest cost incident to node 1. If the optimal 1-tree is a tour, we have found an optimal tour. Otherwise, the degree constraint (6) is violated for one or more nodes. In that case, we proceed by penalizing the degree of such nodes being different from two by perturbing the edge costs of the graph, via the node potentials π : For each edge $(i, j) \in E$, the new edge weight $\tilde{w}(i, j)$ is defined as $\tilde{w}(i, j) = w(i, j) - \pi_i - \pi_j$. Held and Karp [28] show that the optimal TSP tour is invariant under these changes, but the optimal 1-tree is not. One choice for the node potentials is to define $\pi_i = (2 - \deg(i)) \cdot C$, for a constant C (this constant may be updated at each iteration). The Held-Karp procedure re-iterates by solving the 1-tree problem and perturbing the edge costs until it reaches a fixed point or meets a stopping criterion. The best lower bound, i.e., the maximum among all choices of the node potentials, is known as the Held-Karp bound.

Example 1 Consider the undirected weighted graph $G = (V, E, w)$ depicted in Fig. 1.a, i.e., $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, 2), (1, 3), (1, 6), (2, 4), (2, 5), (3, 4), (3, 5), (4, 6), (5, 6)\}$, and edge weights as indicated in the figure (next to each edge). A minimum 1-tree under the edge weights in a. is given in bold in Fig. 1.b. Its objective value is 21. For each node $i \in V$ we update the node potentials as $\pi_i = (2 - \deg(i)) \cdot C$ using $C = 1.5$. The node potentials are indicated in boxes next to the nodes. The corresponding updated edge weights are given in Fig. 1.c., together with a new minimum 1-tree with objective value 24. $\square$

Recall that a minimum spanning tree, and therefore a minimum 1-tree, can be computed in $O(m + n \log n)$ using Prim's algorithm, or in $O(m \log n)$ time using Kruskal's

algorithm [11]. The best known time complexity for computing a minimum spanning tree is $O(m\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function [10].

2.4 Assignment Problem Relaxation

The assignment problem relaxation is obtained by relaxing the connectivity constraints (3) as well as the integrality constraints:

$$\min \sum_{e \in E} w(e)x_e \quad (10)$$

$$\text{s.t. } \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V, \quad (11)$$

$$0 \leq x_e \leq 1 \quad \forall e \in E. \quad (12)$$

This formulation can be applied to symmetric TSPs. However, the assignment problem relaxation is particularly effective on asymmetric TSP instances (see, e.g., [8]). For a directed weighted graph $G = (V, A, w')$ with asymmetric weight function w' , the assignment problem relaxation can be formulated as:

$$\min \sum_{a \in A} w'(a)x_a \quad (13)$$

$$\text{s.t. } \sum_{j \in V, (i,j) \in A} x_{(i,j)} = 1 \quad \forall i \in V, \quad (14)$$

$$\sum_{i \in V, (i,j) \in A} x_{(i,j)} = 1 \quad \forall j \in V, \quad (15)$$

$$0 \leq x_a \leq 1 \quad \forall a \in A. \quad (16)$$

Because the resulting constraint matrix is totally unimodular, this linear programming relaxation will provide an integer optimal solution, if a solution exists. In fact, it can be solved with an efficient combinatorial algorithm, the Hungarian method, which runs in $O(n^3)$ time, and in $O(n^2)$ time when applied incrementally [38, 7].

3 Historical Overview and Related Work

In the first constraint programming system Alice (from 1978), a constraint was introduced to express that a set of variables represent a circuit in a graph [39]. More specifically, to find a circuit covering a set of nodes V , one can define a bijection $C : V \rightarrow V$ representing this circuit as follows in Alice:

FIND BIJ **C** $\rightarrow$ **V V** CIR

where the keywords BIJ and CIR represent ‘bijection’ and ‘circuit’, respectively. That is, for a sequence $v_1, v_2, \dots, v_{|V|}$, we have $C(v_i) = v_{i+1}$ for $i = 1, 2, \dots, |V| - 1$, and $C(v_{|V|}) = v_1$. After several years (in 1994), the constraint appeared in the more modern CP system CHIP, under the name `circuit`, as part of a general extension of the language to include global constraints [4]. Currently, most CP systems include a circuit

constraint, or a similar constraint that is semantically equivalent.¹ Since these circuit constraints are not parametrized with edge weight information, the associated filtering algorithms are based on feasibility only [4, 23]. Furthermore, because a solution to the circuit constraint corresponds to a sequence of variables that take pairwise different values, most CP approaches apply the **alldifferent** constraint [45] in their model (see also Section 4).

We note that the weighted circuit constraint also appeared in the integer programming community, introduced as ‘tour’ in the system SCIL by Althaus et al. [1]. They use this constraint to automatically derive a corresponding integer linear programming model.

Caseau and Laburthe [9] were the first to study cost based filtering algorithms for the weighted circuit constraint. They propose to apply an assignment-based relaxation as well as a spanning tree relaxation, although the main focus is on their assignment model. As a lower bound on the weight of the circuit, they sum the smallest assignments over all variables. They further apply a lazy **alldifferent** propagation, based on the pairwise not-equal constraints. Finally, a separate filtering algorithm is applied that forbids edges that would create a subtour, through a so-called **nocycle** constraint. More precisely, the filtering step of the **nocycle** constraint consists in finding a path of mandatory edges of length at most $n - 1$, and removing the edge between the two endpoints of the path. In other words, they study a decomposition of the weighted circuit constraint, where the cost based filtering is performed on an ad-hoc version of the weighted **alldifferent** constraint, being a weaker version of the assignment problem relaxation of Section 2.4. Caseau and Laburthe also implemented the Held-Karp scheme, that resulted in solving TSPs up to size 70 (i.e., st70 in 3,300 seconds), which takes the TSP structure into account explicitly. They report that the Lagrangean values had to be tuned for each individual instance, and that the approach was therefore not stable and robust enough for application in CP. The work we propose in this paper can be viewed as an extension and improvement of that approach.

The work of Pesant, Gendreau, Potvin, and Rousseau [42] for TSPTW applies **nocycle** constraints to remove edges between two endpoints of a path. They further maintain two lower bounds; one is the ‘greedy’ lower bound by summing the edges with smallest weight incident to each node, similar to the assignment model of Caseau and Laburthe [9]. The other bound is based on the minimum spanning tree. These bounds are used to fathom suboptimal search nodes, but the relaxations are not applied for cost based domain filtering.

The work by Focacci, Lodi, Milano, and Vigo [20] and Focacci, Lodi, and Milano [19, 21, 22] applies (reduced) cost based filtering to optimization constraints, with an application to TSPs. Their methodology allows to apply any (linear) relaxation for this purpose, and they apply the assignment problem relaxation as well as the minimum spanning arborescence relaxation for asymmetric TSPs. The application of the latter relaxation is closely related to our work. The main focus of this sequence of papers, however, is on the assignment problem as a relaxation for the weighted **alldifferent** constraint, which is then applied as a relaxation for the weighted **circuit** constraint in the context of TSPs. This approach is similar to Caseau and Laburthe [9], but Focacci et al. [20] apply the stronger linear programming relaxation. The resulting approach,

¹ We note that one exception is the IBM ILOG constraint programming environment (i.e., CP Optimizer and OPL), that no longer supports the IloPath constraint (to model weighted Hamiltonian paths) nor the **circuit** constraint [33, 34].

using the assignment problem relaxation, is particularly effective on asymmetric TSPs, see for example Focacci et al. [21]. Additional works following this line of research include those by Milano and van Hoesel [41] and Lodi, Milano, and Rousseau [40].

The weighted `circuit` constraint belongs to the more general class of ‘graph constraints’, i.e., constraints that define certain properties on graphs. In particular, the unweighted *tree* constraint [5, 15] can be applied as a relaxation to the `circuit` constraint on directed graphs. Furthermore, the weighted *spanning tree* constraint is closely related to our work [13, 14, 46, 48]. As we will see in Section 5, we can extend several concepts from the weighted spanning tree constraint to handling the 1-tree relaxation for the weighted `circuit` constraint.

The TSP has been studied extensively in the Operations Research literature, including the 1-tree relaxation and the assignment problem relaxation; see [2, 27] for recent overviews. In particular, Grötschel and Holland [26] describe a procedure to solve large-scale symmetric TSPs, an important component of which is the pre-processing phase that removes provably sub-optimal edges based on the reduced costs stemming from the 1-tree relaxation, which is similar to our approach. Furthermore, whenever a new upper bound is found during the solving process, Grötschel and Holland apply standard variable fixing based on the reduced cost stemming from the linear programming relaxation to eliminate sub-optimal edges and fix edges that must appear in an optimal solution. Our edge filtering process is based on the same concept, although we identify the edges to be eliminated or fixed through combinatorial procedures (based on the 1-tree relaxation) rather than a general LP relaxation. Moreover, we embed these procedures inside a domain filtering algorithm and incrementally maintain the data structures throughout the search tree.

State-of-the-art integer programming based TSP solvers (such as Concorde) revolve around a linear programming relaxation of the TSP that contains many specialized cuts that have been developed for the TSP. In addition to subtour elimination constraints, such cuts include comb inequalities, cuts from blossoms, etcetera [2]. As a consequence, these methods must use a generic linear programming solver. During the solving process, the reduced costs associated with this entire linear programming formulation are applied rather than specific reduced costs stemming from, e.g., the 1-tree relaxation. Reasoning based on the reduced costs of the entire LP can be stronger than that of the 1-tree relaxation. We note that instead of removing inconsistent edges, the LP relaxation of Concorde is based on a subset of core edges through a column generation process; edges with negative reduced cost are eligible for inclusion in the LP relaxation. An important restriction, however, is that Concorde can only handle pure symmetric TSPs, as additional side constraints would typically violate the special TSP structure that is utilized in the cut generation algorithms and the upper bound heuristics. In contrast, our combinatorial filtering algorithm is embedded inside a global constraint, which can be naturally combined with additional constraints in a constraint programming system.

4 Representation of the Weighted Circuit Constraint

In the CP literature, different variable representations have been proposed for the (weighted) circuit constraint, the most common of which are the ‘successor’ and the ‘permutation’ representation. We next discuss the consequences of these different representations, and then introduce our set variable representation.

We recall that the *domain* of a variable x , denoted by $D(x)$, is a set of elements that can be assigned to x .

In the *successor* representation, a variable $next_i$ is introduced representing the node that is visited immediately after node i , for all $i \in V$. The initial variable domains can be set to $D(next_i) = V \setminus \{i\}$. Often the successor variables are combined with predecessor variables $pred_i$ representing the node that is visited immediately before node i . Thus, we have $(next_i = j) \Leftrightarrow (pred_j = i)$. In the *permutation* representation, a variable pos_j is introduced representing the j -th node that is visited (pos stands for position), for $j = 1, 2, \dots, n$. That is, pos_1 up to pos_n represent the order in which the nodes are visited. The initial domains can be set to $D(pos_i) = V$.

Both the successor and permutation representation allow the use of **alldifferent** filtering algorithms [45], because one necessary condition for both representations is that the variables take pairwise different values. For the successor representation, additional conditions can be used for filtering purposes, see for example Genç Kaya and Hooker [23]. We note that all CP approaches for the weighted circuit constraint described in Section 3 are based on the successor representation.

The choice of the representation influences the choice of the relaxation that can be applied onto the variables. Namely, for the successor representation (together with the **alldifferent** constraint) the degree constraints cannot be relaxed (i.e., the degree of each node is two), while for the permutation representation the connectivity requirement cannot be relaxed. Therefore, the applicability of a relaxation depends on the choice of variables. For example, the successor representation naturally accommodates the assignment problem relaxation.

We propose a representation that is based on a *set variable* X representing the edges that form the tour.² Using this representation, we can choose to relax either the degree constraints, or connectivity constraints, or both (sequentially). For the domain representation of X , we choose the ‘cardinality+subset’ definition [25, 3].³ That is,

- we require that X has cardinality $|V|$, and
- we maintain a lower and upper bound on the domain of X , where the lower bound $L(X)$ represents all *mandatory* edges, and the upper bound $U(X)$ all *possible* edges.

Initially, we can define $D(X) = [\emptyset, E]$. We also introduce a variable z representing the total weight of the eventual tour represented by X .

In terms of our set variable X , the unweighted **circuit** constraint on a graph $G = (V, E)$ is expressed as **circuit**(X, G), and it states that the edges in X define a Hamiltonian cycle on V in G . For the weighted version, we extend the definition of G with the edge weight function w as $G = (V, E, w)$. The constraint **weighted-circuit**(X, z, G) states that the edges in X form a Hamiltonian cycle on the nodes V in G with weight at most z , i.e.,

$$\text{weighted-circuit}(X, z, G) = \text{circuit}(X, G) \wedge \left(\sum_{e \in X} c(e) \leq z \right).$$

During the search for a solution, the domains of X and z will change as a result of constraint propagation caused by constraints other than **weighted-circuit**. Therefore,

² Set variables were introduced independently by Puget [44] and Gervet [24].

³ We note that alternative set domain representations exist, but for the purpose of our paper, the cardinality+subset representation is most natural. Moreover, it is available in most existing CP systems.

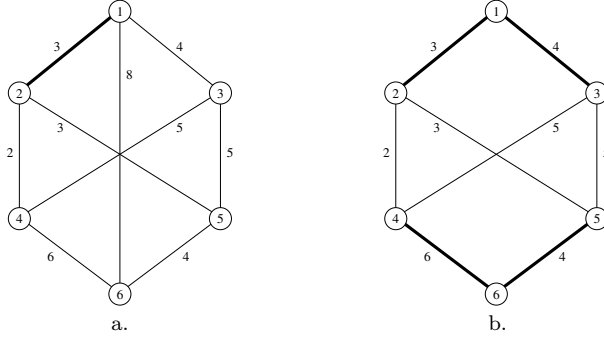


Fig. 2 a. The graph corresponding to Example 2. Edge weights are indicated next to each edge. Bold edges are mandatory. b. Graph representation reflecting the domain changes after filtering.

given domains for X and z , domain filtering for **weighted-circuit** amounts to the following tasks:

- i) Identify edges in G that *must* be part of a solution and add these to the lower bound $L(X)$ (the mandatory edges),
- ii) Identify edges in G that *cannot* be part of a solution and remove these from the upper bound $U(X)$ (the possible edges),
- iii) Identify the minimum value that z can take and use this to increase its lower bound.

We also refer to edges that cannot be part of a solution as *inconsistent edges*. When a filtering algorithm removes all inconsistent edges from $U(X)$ and adds all mandatory edges to $L(X)$, it is said to establish *bounds consistency* on **weighted-circuit** [25]. Because finding even a single solution to the **weighted-circuit** constraint is already an NP-complete problem [36], establishing bounds consistency is NP-hard. Therefore, we will focus on developing effective, and computationally efficient, filtering algorithms for the **weighted-circuit** constraint that do not necessarily establish bounds consistency.

Example 2 Consider again the undirected weighted graph $G = (V, E, w)$ from Example 1, with the original edge weights as in Fig. 1.a. Furthermore, consider set variable X with domain $D(X) = [\{(1, 2)\}, E]$, and variable z with interval domain $D(z) = [0, 25]$. Fig. 2.a depicts G again, where the mandatory edge of X is indicated in bold.

For illustrative purposes we next identify all four possible circuits that include mandatory edge $(1, 2)$, and their corresponding weight:

- 1) 1-2-4-6-5-3-1 with weight 24
- 2) 1-2-5-3-4-6-1 with weight 30
- 3) 1-2-5-6-4-3-1 with weight 25
- 4) 1-6-5-3-4-2-1 with weight 27

Let us analyze the constraint **weighted-circuit**(X, z, G). First, observe that all edges in the graph can be part of a circuit that contains the mandatory edge of X , and moreover, no edge (other than the mandatory edge) appears in all circuits. Thus, based on feasibility alone, we cannot deduce any further information.

Taking into account the weights, however, we can identify that edge $(1, 6)$ appears only in solutions with weight at least 27, while z allows solutions with weight at most 25. Therefore, we can remove edge $(1, 6)$ from the upper bound of X . In fact, we

can inspect the two feasible solutions 1) and 3) and identify that edges (1, 3), (4, 6), (5, 6) always appear in a solution. They can therefore be added to the lower bound (mandatory edges) of X . Finally, we can also increase the lower bound of z to 24; the minimum weight over all solutions. $\square$

5 Filtering for the Weighted Circuit Constraint

As mentioned before, we propose filtering algorithms based on relaxations of **weighted-circuit**, where our primary focus will be on the 1-tree relaxation that was introduced by Held and Karp. Additionally, we apply combinatorial filtering based on connectivity requirements. The approach we take for identifying mandatory and possible edges is closely related to that of the weighted spanning tree constraint [14, 46, 48]. Namely, by definition a 1-tree on a graph G contains a spanning tree on the graph $G \setminus \{1\}$. We therefore adapt the concepts and algorithms that were developed for the minimum spanning tree to the graph $G \setminus \{1\}$.

Throughout this section, we let $G = (V, E, w)$ be an undirected weighted graph. We let $T(G)$ denote a minimum 1-tree of G . We let $T_e(G)$ denote a minimum 1-tree of G containing edge $e \in E$. For a subset of edges $E' \subseteq E$, we let $w(E')$ denote $\sum_{e \in E'} w_e$.

5.1 Removing Edges Based on Marginal Costs

The *marginal cost* of an edge $e \in E$ with respect to $T(G)$ is defined as $\bar{w}(e) = w(T_e(G)) - w(T(G))$. That is, it represents the marginal weight increase if e is forced into the minimum 1-tree. If $T_e(G)$ does not exist (for example because e forms a cycle with mandatory edges in $G \setminus \{1\}$), we define $\bar{w}(e) = \infty$. We note that the term ‘marginal cost’ is commonly used in the context of sensitivity analysis for minimum spanning trees [51, 50, 12]. We will apply marginal costs to identifying inconsistent edges, based on the following lemma.

Lemma 1 *Consider the constraint **weighted-circuit**(X, z, G) where $G = (V, E, w)$, and let $e \in E$. If*

$$w(T(G)) + \bar{w}(e) > \max(D(z)),$$

then e is inconsistent and can be removed from $U(X)$.

Proof An edge is inconsistent if $w(T_e(G)) > \max(D(z))$, and $w(T_e(G)) = w(T(G)) + \bar{w}(e)$ by definition. $\square$

Observe that Lemma 1 is invariant under the application of node potentials π to the edge weights of G , as discussed in Section 2. In particular, each 1-tree that is computed during the Held-Karp process can be used for filtering purposes. This allows, for example, to terminate the Held-Karp process once a certain time or quality criterion has been met, and safely apply the resulting relaxation. We remark that this property can be extended to more general Lagrangean-based filtering; see Sellmann [49].

In order to apply Lemma 1 to filter inconsistent edges, we need to compute the marginal cost for each edge. Computing $\bar{w}(e)$ via $T_e(G)$ for each individual edge $e \in E$ would take $O(m^2 \alpha(m, n))$ time in the worst case, which is not practical. We next describe how we can compute the marginal cost for all edges in $G \setminus \{1\}$ more efficiently, given a 1-tree.

Lemma 2 Let $G = (V, E, w)$ be a weighted graph and let $T(G)$ be a minimum 1-tree of G . Let $e = (i, j) \in E$ such that $e \notin T(G)$ and $i, j \neq 1$. Let $P_{i,j}$ be the unique i - j path in $T(G) \setminus \{1\}$. Then

$$\bar{w}(e) = w(e) - \max\{w(a) \mid a \in P_{i,j}\}.$$

Proof By definition, a minimum 1-tree consists of a minimum spanning tree M on $G \setminus \{1\}$ and two smallest-weight edges incident to node 1. The edges $e = (i, j) \in E$ such that $i, j \neq 1$ are precisely all edges in $G \setminus \{1\}$. Therefore, for all edges $e = (i, j) \in E$ such that $e \notin T(G)$ and $i, j \neq 1$, $\bar{w}(e)$ is equivalent to the marginal cost of e with respect to M . The result then follows from, e.g., [51]. $\square$

Finding the unique path $P_{i,j}$ can be achieved through a depth-first search on $T(G)$ in $O(n)$ time for an edge (i, j) . This immediately yields an overall $O(mn)$ time complexity for computing $\bar{w}(e)$ for all $e \in E$ such that $e \notin T(G)$, $e \notin \delta(1)$.

We recall that very similar filtering has been applied to the weighted spanning tree constraint by [14, 46]. In particular, we can readily apply the algorithm of [46] and obtain an improved overall time complexity of $O(n + m + n \log n)$.

Lastly, the marginal cost for the edges that are incident to node 1 can be computed as follows.

Lemma 3 Let $G = (V, E, w)$ be a weighted graph and let $T(G)$ be a minimum 1-tree of G . Let $e \in \delta(1)$ such that $e \notin T(G)$. Then

$$\bar{w}(e) = w(e) - \max\{w(a) \mid a \in \delta(1), a \in T(G)\}.$$

Proof Computing $\bar{w}(e)$ via $T_e(G)$ can be done by adding $\max_{a \in E} w(a)$ to $w(e')$ for all edges $e' \in \delta(1)$, $e' \neq e$ and finding the minimum 1-tree in the modified graph. This 1-tree then includes edge e and $\arg\min\{w(a) \mid a \in \delta(1), a \in T(G)\}$ to be incident to node 1, effectively replacing $\arg\max\{w(a) \mid a \in \delta(1), a \in T(G)\}$ with e . $\square$

Example 3 Consider again graph G from Example 1, and the constraint **weighted-circuit** (X, z, G) , with $D(X) = [\emptyset, E]$ and $D(z) = [0, 25]$. We apply the updated edge weights and the minimum 1-tree from Fig. 1.c. to filter the constraint.

First, consider edge $(2, 5)$ with weight 5. Applying Lemma 2, the path $P_{2,5}$ is 2-4-6-5, with a maximum weight edge of value 4.5 (i.e., edge $(4, 6)$). Therefore, the marginal cost $\bar{w}(2, 5) = 5 - 4.5 = 0.5$. Recall that the 1-tree has objective value 24, while the upper bound for z is 25. Therefore we cannot remove $(2, 5)$ from $U(X)$. Consider next edge $(1, 6)$ with weight 6.5. By Lemma 3, its marginal cost is $\bar{w}(1, 6) = 6.5 - 4.5 = 2$. Since $24 + 2 > 25$, we can remove $(1, 6)$ from $U(X)$ by Lemma 1. $\square$

Let us next compare our filtering based on marginal costs with the feasibility filtering of Caseau and Laburthe [9] and Pesant et al. [42] based on the **nocycle** constraint. Recall that the filtering step of the **nocycle** constraint consists in finding a path of mandatory edges of length at most $n - 1$, and removing the edge between the two endpoints of the path. Using our framework, that removed edge would have a marginal cost determined by the unique path in the tree, that is formed by mandatory edges only. In such case, the replacement cost is infinite, and hence the edge will be removed also in our framework. Therefore, our algorithm is stronger than the feasibility filtering of [9] and [42] based on the **nocycle** constraint.

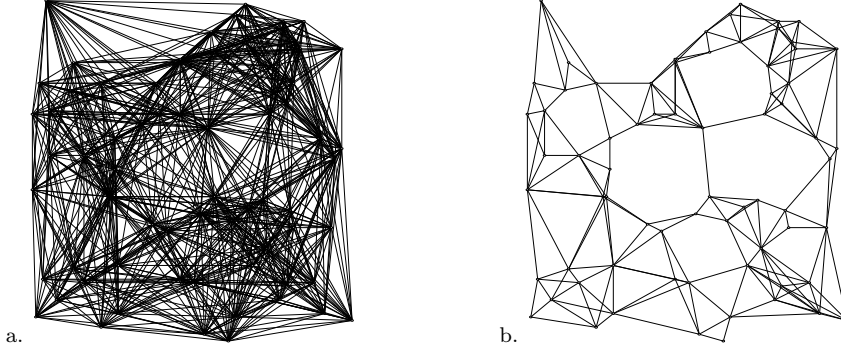


Fig. 3 The filtered graph for **st70** with respect to an upper bound of 700 (a) and 675 (b).

Fig. 3 presents an illustration of the practical impact of our edge filtering based on marginal costs, applied to the complete graph **st70** from TSPLIB. In Fig. 3.a we show the graph after removing the filtered edges with respect to an upper bound of 700, while Fig. 3.b shows the effect of edge filtering with respect to an upper bound of 675 (the length of the optimal tour, which was heuristically found).

5.2 Identifying Mandatory Edges

The *replacement cost* of an edge $e \in T(G)$ is defined as $w'(e) = w(T(G \setminus \{e\})) - w(T(G))$, where we define $w(T(G \setminus \{e\})) = \infty$ if $G \setminus \{e\}$ is not connected. It represents the weight increase of the minimum 1-tree when a tree edge e is removed from G , and hence must be replaced with a non-tree edge. We can apply replacement costs to identify which edges are mandatory, using the following immediate lemma.

Lemma 4 Consider the constraint **weighted-circuit**(X, z, G) where $G = (V, E, w)$. Let $T(G)$ be a minimum 1-tree of G and let $e \in T(G)$. If $w(T(G)) \leq \max(D(z))$ and

$$w(T(G)) + w'(e) > \max(D(z)),$$

then e is mandatory and can be added to $L(X)$.

Similar to the marginal costs for non-1-tree edges, we separate the computation of the replacement costs for 1-tree edges in $G \setminus \{1\}$ and those in $\delta(1)$.

Lemma 5 Let $G = (V, E, w)$ be a weighted graph and let $T(G)$ be a minimum 1-tree of G . Let $e \in T(G) \setminus \{1\}$. Let T^1 and T^2 be the two sub-trees that form $(T(G) \setminus \{1\}) \setminus \{e\}$. Then

$$w'(e) = w(e) - \min\{w(i, j) \mid (i, j) \in E, (i, j) \neq e, i \in T^1, j \in T^2\}.$$

Proof Analogous to Lemma 2, $w'(e)$ is equivalent to the replacement cost of e in the minimum spanning tree of $G \setminus \{1\}$. The result then follows from, e.g., [51]. $\square$

Example 4 We continue Example 3. Consider edge $(4, 6)$ with weight 4.5. Removing this edge from $T(G) \setminus \{1\}$ yields two subtrees with respective node sets $\{2, 3, 4\}$ and $\{5, 6\}$. We can minimally reconnect these sets through edge $(2, 5)$ with weight 4.5. Therefore, the replacement cost of $(4, 6)$ is $w'(4, 6) = 4.5 - 4.5 = 0$ by Lemma 5, and we cannot conclude that this edge is mandatory. $\square$

Lemma 5 gives rise to the following algorithm for computing the replacement costs in $G \setminus \{1\}$. First, we order all non-1-tree edges by non-increasing weight. We then consider each of these edges in turn. Non-tree edge (i, j) forms a (unique) circuit with edges in $T(G) \setminus \{1\}$. The first edge that we consider (i.e., with smallest weight) serves as ‘replacement edge’ for all edges on the unique circuit. Similarly, the non-tree edges that are considered subsequently will serve as replacement edge for all edges on their respective circuit that have not yet been assigned a replacement edge. As each circuit may be of length $O(n)$, this algorithm has an $O(mn)$ time complexity.

The theoretically best known time complexity for finding all edge replacement costs for minimum spanning trees is $O(m\alpha(m, n))$ [51, 12], and we immediately inherit this time complexity for computing the replacement costs of the edges in $T(G) \setminus \{1\}$. In fact, these ideas have been applied before to identify mandatory edges for the weighted spanning tree constraint, by [14, 48]. In particular, we can apply the algorithm of [48] that achieves the time complexity of $O(m\alpha(m, n))$, in a manner that is practically suitable for constraint programming systems.

Lastly, the replacement cost for the edges that are incident to node 1 can be computed as follows, analogous to Lemma 3.

Lemma 6 *Let $G = (V, E, w)$ be a weighted graph and let $T(G)$ be a minimum 1-tree of G . Let $e \in \delta(1)$ such that $e \in T(G)$. Then*

$$w'(e) = w(e) - \min\{w(a) \mid a \in \delta(1), a \notin T(G)\}.$$

5.3 Forcing Edges Based on Degree Constraints

We next discuss an additional filtering rule that follows from connectivity considerations, and can be applied at no extra cost during the computation of the 1-tree using Prim’s algorithm. First, observe that the connectivity constraints (3) in the TSP model of Section 2 can alternatively be modeled as (see, e.g., [2]):

$$\sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, |S| \geq 3. \quad (17)$$

A useful consequence of (17) is the following corollary.

Corollary 1 *Consider the constraint **weighted-circuit**(X, z, G) where $G = (V, E, w)$. Let S be a proper nonempty subset of V . If $|\delta(S)| = 2$, then both edges in $\delta(S)$ are mandatory and can be added to $L(X)$.*

Recall that Prim’s algorithm [43] computes the minimum spanning tree in $G \setminus \{1\}$ in the following manner. Starting from any node $v \in V$, it first partitions V into disjoint subsets $S = \{v\}$ and $\bar{S} = V \setminus \{v\}$ and creates an empty tree T . Then it iteratively adds to T the minimum edge $(i, j) \in \delta(S)$ such that $i \in S$, and moves j from $\bar{S}$ to S . We can apply Corollary 1 at each step during Prim’s algorithm: Whenever we encounter a partition $(S, \bar{S})$ such that $\delta(S)$ contains two edges, we can add these to the set of mandatory edges.

Example 5 We continue Example 3, that is, with edge $(1, 6)$ removed from $U(X)$. Assume that Prim’s algorithm starts with node 6, i.e., $S = \{6\}$ and $\bar{S} = \{2, 3, 4, 5\}$. We have $\delta(S) = \{(4, 6), (5, 6)\}$ with cardinality 2. Hence, by Corollary 1 both edges $(4, 6)$ and $(5, 6)$ are mandatory. $\square$

5.4 Handling Asymmetric Instances

In Section 4, we have made no assumption on G to be directed or not. The filtering algorithms presented in this section, however, were based on the assumption that G is undirected. In case G is a directed graph with asymmetric edge weights, there are two possibilities to filter the **weighted-circuit** constraint. The first is to apply a directed version of the 1-tree relaxation (referred to as 1-arborescence in Held and Karp [28]), and adapt the filtering algorithms accordingly. The second possibility is to convert the asymmetric instance into a symmetric one [35], to which we apply the techniques for the 1-tree relaxation. Our initial computational experiments showed that the 1-arborescence performs worse than the 1-tree relaxation on the transformed instance, both in terms of solution quality (lower bound) and algorithmic stability. Therefore, we adopted the second possibility in our implementation and experimental results.

We next describe the method of Jonker and Volgenant [35] for transforming an asymmetric instance into a symmetric one, as we will need these details in the following section on additive bounding. Let $G = (V, A, w)$ be a directed weighted graph with node set $V = \{1, 2, \dots, n\}$, arc set A , and asymmetric weight function w . Let C be a $n \times n$ weight matrix defined such that for $i, j \in V$, $C_{i,j} = w(i, j)$ if $(i, j) \in A$, $C_{i,j} = \infty$ if $(i, j) \notin A$ and $i \neq j$, and $C_{i,j} = -M$ if $i = j$, where M is a very large number. In other words, C exactly represents the weight function w , except that its diagonal entries are $-M$. We also define the $n \times n$ matrix W as $W_{i,j} = \infty$ for all $i, j \in V$.

The asymmetric instance is now transformed into a symmetric instance on a complete graph with $2n$ nodes and weight matrix $\tilde{C}$ defined as

$$\tilde{C} = \begin{bmatrix} W & C^\top \\ C & W \end{bmatrix},$$

that is, the edge weight function on the symmetric instance returns $\tilde{C}_{i,j}$ for an edge (i, j) , where $i, j \in \{1, 2, \dots, 2n\}$.

Assuming that the original instance has a bounded solution, the optimal solutions of the new instance are bounded as well, and contain exactly n edges of weight $-M$. Solutions occur in pairs, and one pair takes the form

$$i_1 \rightarrow (i_1 + n) \rightarrow i_2 \rightarrow (i_2 + n) \rightarrow \dots \rightarrow i_n \rightarrow (i_n + n) \rightarrow i_1,$$

with $i_k \in V$ for $k = 1, 2, \dots, n$. These solutions occur in 1-to-1 correspondence with the optimal solutions to the original asymmetric instance: One can delete nodes with index greater than n from the solution, and add nM to its objective value to obtain the corresponding solution to the original problem.

Recall that since $\tilde{C}$ is symmetric, we only need to represent one undirected edge for each pair (i, j) and (j, i) of $\tilde{G}$, with bounded weight. We choose to represent the lower diagonal edges $\tilde{C}$, i.e., we let arc $(i, j) \in A$ be represented by edge $(i + n, j)$ in E .

6 Improved Filtering Through Additive Bounding

Fischetti and Toth [16] introduced an *additive bounding* procedure that combines different relaxations to obtain a valid lower bound for a given problem. More precisely,

following the description of [16], the additive bounding procedure takes as input a problem P of the form

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & x \in F(P), \end{aligned}$$

where x and c are, respectively, a column vector of variables and a cost row vector, both having n elements, and $F(P) \subset \{x \in \mathbb{R}^n : x \geq 0\}$.

Let $L^{(1)}, \dots, L^{(r)}$ be lower bounding relaxations available for problem P . Assume that, for $i = 1, \dots, r$, lower bounding procedure $L^{(i)}(\bar{c})$ —when applied to problem P with cost vector $\bar{c}$ —returns a lower bound value $v^{(i)}$ as well as a residual cost row vector $c^{(i)} \in \mathbb{R}^n$ such that

$$c^{(i)} \geq 0, \text{ and} \tag{18}$$

$$v^{(i)} + c^{(i)}x \leq \bar{c}x \text{ for each } x \in F(P). \tag{19}$$

For example, a linear programming relaxation for P will produce in addition to a lower bound also a vector of reduced costs, that fit precisely the above definition of residual cost. The additive bounding procedure starts with applying relaxation $L^{(1)}$ to the original cost vector c . Then, for $i = 2, \dots, r$ it will sequentially apply relaxation $L^{(i)}(c^{(i-1)})$. Fischetti and Toth show that $v^{(1)} + v^{(2)} + \dots + v^{(r)}$ is a valid lower bound for P .

Additive bounding has been used in the context of constraint programming to obtain stronger domain filtering for specific problem structures, in particular *discrepancy-based* additive bounding [18, 41, 40].

In our framework, using a set variable representation, we can easily integrate the 1-tree relaxation and the assignment problem relaxation using additive bounding. As these relaxations represent complementary structures of the **weighted-circuit** constraint, together they may yield a stronger additive bound. More precisely, we propose to use the marginal costs associated with the 1-tree relaxation to define residual costs for the assignment problem relaxation. For symmetric instances, the 1-tree and assignment problem are defined on the same graph, and we can immediately define the residual cost of an edge e as the marginal cost $\bar{w}(e)$ stemming from the 1-tree relaxation. For asymmetric instances, we take into account the transformation described in Section 5.4, as follows. Let $G = (V, A, w)$ be the original directed graph, and let $\tilde{G} = (\tilde{V}, E, \tilde{w})$ be the undirected graph after applying the transformation. We first apply the 1-tree relaxation to $\tilde{G}$. We then apply the Assignment Problem relaxation to the original graph, where the residual cost of arc $(i, j) \in A$ is made equal to the marginal cost of edge $(i + n, j)$ in E .

Lastly, we show that:

Lemma 7 *Marginal costs are valid residual costs for the additive bounding framework.*

Proof By definition marginal costs are always positive, which proves condition (18). In terms of our application, the second condition (19) is stated as

$$w(T(G)) + \sum_{e \in E} \bar{w}(e)x_e \leq \sum_{e \in E} w(e)x_e. \tag{20}$$

Before we prove this, for each edge $e \in E$ we denote by e^r the ‘replacement edge’ of e with respect to $T(G)$. That is, from Lemma 2, for an edge $e = (i, j)$ we have $e^r = \operatorname{argmax}\{w(a) \mid a \in P_{i,j}\}$ where $P_{i,j}$ is the unique i - j path in $T(G) \setminus \{1\}$. We

will use this to rewrite $w(T_e(G))$ as $w(T(G)) + w(e) - w(e^r)$. Statement (20) can be derived as

$$\begin{aligned}
 w(T(G)) + \sum_{e \in E} \overline{w}(e)x_e &= w(T(G)) + \sum_{e \in E} (w(T_e(G)) - w(T(G)))x_e \\
 &= w(T(G)) + \sum_{e \in E} (w(T(G)) + w(e) - w(e^r) - w(T(G)))x_e \\
 &= w(T(G)) - \sum_{e \in E} w(e^r)x_e + \sum_{e \in E} w(e)x_e \leq \sum_{e \in E} w(e)x_e,
 \end{aligned}$$

where the last inequality follows from $w(T(G)) - \sum_{e \in E} w(e^r)x_e \leq 0$ as $w(T(G))$ is the value of a minimum 1-tree. $\square$

7 Computational Results

We next evaluate our proposed algorithms experimentally. Our main focus will be on assessing the impact of domain filtering, with respect to using only the bounds from the relaxations. We will measure this for the 1-tree relaxation, and compare it to the assignment problem relaxation and the combined additive bounding relaxation. As test problem domain, we use symmetric and asymmetric TSP instances. We use structured instances from TSPLIB as well as randomly generated instances. As we are interested here in the effect of filtering, we provide each method with the same upper bound that we obtained using the state-of-the-art Lin-Kernighan-Helsgaun algorithm [30, 31].

Additionally, we compare our methodology to the TSP solver Concorde. The main reason for this is to analyze the respective strengths and weaknesses of our method and this dedicated solving method. In particular, we would like to validate how much slower our CP approach is with respect to this special-purpose TSP solver. Furthermore, we would like to measure the possible amount of overhead generated by the linear programming solver used in Concorde with respect to our, more light-weight, algorithms. A typical measure in this context is the number of search nodes that are generated per second, from which the average time spent per search node can be derived. Namely, if two approaches provide the same amount of total solving time for the **weighted-circuit** constraint alone, a CP solver would generally favor the approach that generates the most number of search nodes, to allow more effective interaction with other problem constraints through constraint propagation in more complex models.

Finally, we would like to compare our methodology to existing state-of-the-art CP technology. Unfortunately, none of the major CP solvers currently provides a filtering algorithm for the **weighted-circuit** constraint. We therefore selected IBM ILOG CP Optimizer, which is among the fastest CP solvers currently available, to compare our method with. We applied the following model (as suggested in the documentation of CP Optimizer), that combines the successor and the permutation representation. Let $G = (V, E, w)$ be the weighted graph under consideration, with $|V| = n$. For each $i \in V$ we introduce variables $next_i$, and for each $j \in \{1, 2, \dots, n\}$, we introduce a variable

pos_j (as in Section 4). We then formulate the problem as

$$\begin{aligned}
& \min \sum_{i \in V} w(i, next_i) \\
& \text{s.t. } \text{alldifferent}(next_1, \dots, next_n) \\
& \quad \text{alldifferent}(pos_1, \dots, pos_n) \\
& \quad pos_j = next_{pos_{j-1}} \quad \forall j \in \{2, \dots, n\} \\
& \quad pos_1 = 1
\end{aligned}$$

For symmetric problem instances, we break symmetry (reverse tours) by additionally introducing variables $pred_i$ for each $i \in V$, and introducing the constraints

$$\text{alldifferent}(pred_1, \dots, pred_n) \quad (21)$$

$$\text{inverse}(next, pred) \quad (22)$$

$$next_1 < pred_1 \quad (23)$$

Here, the **inverse** constraint in (22) represents the relationships $(next_i = j) \Leftrightarrow (pred_j = i)$ for all pairs i, j ($i \neq j$), while constraint (23) forbids the reverse tours.

Since the default search of CP Optimizer did not perform well, we apply a depth-first search to this model over the $next$ variables, using minimum domain size as variable ordering heuristic, and the minimum distance value as value ordering heuristic. We use ‘extended filtering’ as inference level for the constraints. The comparison with this model will give insight in the additional strength that filtering algorithms for the **weighted-circuit** can provide to existing CP technology.

7.1 Implementation Details

We have implemented all proposed algorithms in C++: The 1-tree relaxation, the assignment problem relaxation, the filtering algorithms, the additive bounding procedure, and a standard branch-and-bound search procedure. For our experiments, we have used two different branching schemes. The first is called ‘binary branching’: At each branch and bound node, choose the edge e with maximum replacement cost and create two children nodes, one where e is removed, and one with e forced into the solution. The second branching scheme is called ‘tree branching’: At each branch and bound node, we first sort the edges by non-increasing replacement cost. We create multiple children nodes as follows. In the first child node, the first edge (i.e., with maximum replacement cost) is removed. In the second child node, the first edge is forced and the second edge is removed. In the third child node, the first two edges are forced and the third is removed, and so on. The binary branching scheme is more effective on structured problem instances from TSPLIB, while the tree branching scheme is more suitable for randomly generated instances.

As stated before, the proposed filtering algorithms do not establish bounds consistency on the **weighted-circuit** constraint. Moreover the algorithms are not idempotent, which means that repeated application of the algorithms may filter (remove or force) more edges. Therefore, we investigated the effect of increasing the repeated application of our filtering algorithms until we reach a fixpoint. Table 1 summarizes an experimental evaluation on TSPLIB instances (we refer to [48] for more details). We report the average solving time and number of search tree nodes, as well as the average number of search nodes that is explored per second. Observe that filtering until

	no filtering	one round	fixpoint
average time (s)	693.64	1.66	3.38
average search nodes	9,833.80	489.28	453.60
average nodes/s	115.52	177.38	136.90

Table 1 Comparing the 1-tree relaxation without filtering (no filtering) with one round of filtering (one round), and filtering until a fixpoint is reached (fixpoint), on TSPLIB instances.

the fixpoint can reduce the number of search nodes, but the computational overhead does not warrant the limited additional filtering of the fixpoint computation. In the experiments that follow, we therefore only apply the filtering algorithms once at each search node.

Our assignment problem implementation closely follows the approach of Focacci, Lodi, Milano, and Vigo [20] and Focacci, Lodi, and Milano [19]. That is, we implemented the Hungarian method (a combinatorial algorithm for solving the assignment problem), and applied reduced cost based filtering. When we run our branch and bound procedure solely with the assignment problem relaxation, we apply the binary branching scheme.

The results with the Concorde solver [2] were obtained by using version Concorde-03.12.19 together with IBM ILOG CPLEX 12.2 as linear programming solver. We also report results for IBM ILOG CP Optimizer 12.2 (as part of the CPLEX 12.2 Optimization Studio). We used the Lin-Kernighan-Helsgaun algorithm, version LKH-2.0.5, to compute upper bounds to the traveling salesman problems. All experiments are performed using a 2.33GHz Intel Xeon machine with 8GB memory.

7.2 Symmetric Traveling Salesman Problems

In our first set of experiments we randomly generated TSP instances on graphs with 50 to 550 nodes (with increments of 50). That is, for a given number of nodes n , we generate an $n \times n$ weight matrix where each upper-diagonal entry is a uniform-randomly generated integer between 1 and 1,000, corresponding to the ‘class A’ type problems described in [17]. For each number of nodes, we generated 30 instances. Each of these instances was solved by 1) our branch-and-bound solver using the 1-tree relaxation but no filtering, 2) our branch-and-bound solver using the 1-tree relaxation and filtering, and 3) the solver Concorde. Our branch-and-bound solver uses the tree branching scheme as search strategy, and was run with a time limit of 1,800 seconds.

In Fig. 4 we present a log-log scatter plot that compares the impact of our filtering algorithms to the branch-and-bound search, in terms of search tree nodes (a) and solving time (b). We note that whenever a method reaches the time limit for an instance, we report this as 10^4 in Fig. 4.a for visualization purposes (10^4 was the maximum number of search nodes reported for an instance). That is, for such instances, the actual number of search nodes explored within 1,800 seconds may be much lower. Fig. 4.a clearly illustrates the impact of filtering, in that the number of search tree nodes can consistently be dramatically reduced. This is also reflected in the running time, and an interesting observation from Fig. 4.b is that the relative performance of the filtering algorithms increases for larger instances. We attribute this to the fact that our filtering algorithms not only reduce the search tree size, but also impact the running times of

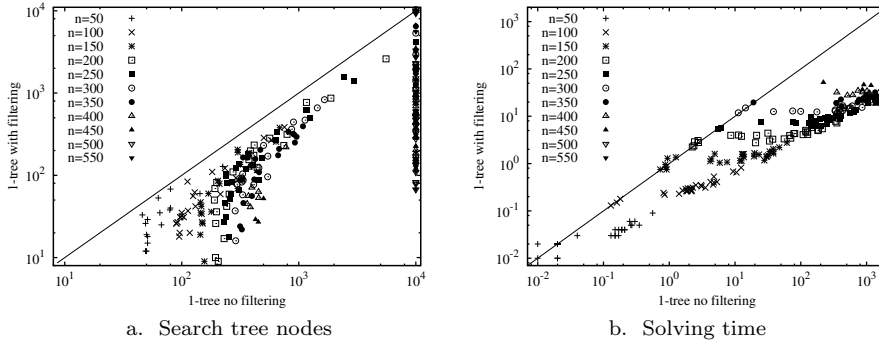


Fig. 4 Comparing the 1-tree relaxation with and without filtering on randomly generated symmetric TSP instances, in terms of number of search tree nodes (a) and solving time in seconds (b). The solving time limit was set to 1,800 seconds. Each data point in these figures corresponds to an instance.

size	1-tree no filtering			1-tree with filtering			Concorde		
	solved	time	nodes/s	solved	time	nodes/s	solved	time	nodes/s
50	1.00	0.13	299.26	1.00	0.03	712.39	1.00	0.18	19.59
100	1.00	3.19	55.10	1.00	0.34	160.65	1.00	0.31	6.10
150	1.00	18.31	13.83	1.00	1.42	46.91	1.00	0.59	4.52
200	1.00	132.30	5.16	1.00	4.68	33.00	1.00	0.97	3.18
250	0.97	409.88	2.13	1.00	10.98	25.76	1.00	1.98	2.83
300	0.80	770.67	1.38	1.00	24.35	20.29	1.00	2.32	2.15
350	0.67	1,239.25	0.61	1.00	39.54	15.96	1.00	3.74	1.92
400	0.33	1,589.71	0.42	0.97	108.45	11.04	1.00	4.57	1.64
450	0.17	1,722.56	0.34	1.00	121.08	12.16	1.00	4.99	1.68
500	0.00	1,800.00	0.21	0.97	194.32	8.81	1.00	6.42	1.38
550	0.00	1,800.00	0.20	0.97	206.99	7.98	1.00	5.00	1.00

Table 2 Comparing the 1-tree relaxation with and without filtering, and with Concorde, on the same randomly generated symmetric TSP instances as were used in Fig. 4. For each method and size class, we report the fraction of instances solved, average solving time, and average number of search nodes processed per second. All reported times are in seconds; the solving time limit was set to 1,800 seconds.

these algorithms. Most importantly, the computation of a 1-tree takes much less time on a filtered graph.

In Table 2 we report these result on these same instances in an aggregated way, by averaging instances of the same size. For each method (1-tree no filtering, 1-tree with filtering, and Concorde), and each problem size, we report the fraction of instances solved to optimality, the average solving time (by taking 1,800 seconds for instances that reach the time limit this number serves as a lower bound), and the average number of nodes per second processed in the branch-and-bound search. When using only the 1-tree relaxation, we can handle instances up to 250 nodes. When we add our filtering algorithms, we can handle almost all instances up to 550 nodes. Naturally, the dedicated solver Concorde can also handle all these problems easily. However, Table 2 presents two interesting observations. First, for small to medium sized problems (up to around 150 nodes), the 1-tree relaxation with filtering is competitive to Concorde, and sometimes even faster. Second, the number of nodes processed per second is much larger for the 1-tree relaxation with filtering than for Concorde. This indicates that our light-weight

filtering algorithms can be much more attractive to embed in a generic CP solver than the heavier linear programming relaxation that is used in Concorde.

We next report results on a selection of TSPLIB instances with less than 300 nodes, that could be optimally solved within the time limit by one of our branch-and-bound algorithms. The results are reported in Table 3, that shows for each method (IBM ILOG CP Optimizer, 1-tree no filtering, 1-tree with filtering, and Concorde) the best found solution, the number of search nodes, and the solving time in seconds. For these instances, our branch-and-bound solver uses the binary branching scheme as search strategy. All methods were run with a time limit of 1,800 seconds. The upper bound is given to each method without a witness, and happens to correspond to the optimal value for all reported instances. Hence, the task for each method is to find the optimal tour, and prove its optimality. Our branch and bound methods, however, also keeps track of the best tour found during search (for example during the computation of a 1-tree). In some cases, the value of this best tour may exceed the given upper bound, but we still report it as ‘best tour’ in the table.

The first observation from Table 3 is that the **weighted-circuit** constraint with only the 1-tree relaxation already greatly outperforms the standard CP model. The performance of the **weighted-circuit** is further improved when the filtering algorithms are also active. A second observation is that for small to medium-sized problems (including all TSPLIB instances up to **e1176**), our method is competitive to Concorde. This again indicates that our filtering algorithms are relatively powerful already. Moreover, these results are particularly relevant because in most practical vehicle routing problems, the tours are of small to medium size.

Lastly, we mention that previous CP-based methods for TSPLIB instances did not scale beyond small sized instances: The hybrid method of Focacci et al. [22], using the assignment problem relaxation, can solve instance **gr48**, while an extension of that method presented in [41] (using discrepancy-based subproblem generation) can solve instance **brazil58**. The largest TSPLIB instance solved so far by a CP method was **st70**, by employing a 1-tree relaxation [9]. Our filtering algorithms for **weighted-circuit** thus improve on the state of the art in CP methods for solving TSP instances, and show that filtering based on the 1-tree relaxation can be performed in a stable and robust enough manner.

7.3 Asymmetric Traveling Salesman Problems

We next present results on asymmetric TSP instances. All our algorithms presented in this section apply the binary branching scheme. We first validate the performance of our 1-tree filtering algorithm on random asymmetric problem instances, similar to the random symmetric instances above (that is, using ‘type A’ instances from [17]). The results are presented in Fig. 5, which follows the same format as Fig. 4. Based on these figures, we observe that the relative performance of our 1-tree filtering algorithms is similar for symmetric and asymmetric instances.

We also compare the three different relaxations (1-tree, assignment problem, and the additive bound) on these randomly generated instances. We remark that the assignment problem relaxation is typically more effective on asymmetric instances than on symmetric ones [17], which makes these instances particularly appropriate as a benchmark set. The results are reported in Table 4, which follows the same format as Table 2. These results indicate that generally the assignment problem relaxation

instance	UB	IBM ILOG CP Optimizer			1-tree no filtering			1-tree with filtering			Concorde		
		best found	search nodes	time	best found	search nodes	time	best found	search nodes	time	best found	search nodes	time
burma14	3323	3323	9,455	0.76	3323	1	0.01	3323	1	0.01	3323	1	0.02
ulysses16	6859	6859	62,789	5.13	6859	1	0.01	6859	1	0.00	6859	1	0.05
gr17	2085	2085	608,220	66.34	2085	1	0.01	2085	1	0.01	2085	1	0.02
gr21	2707	2707	8,516	1.65	2707	1	0.00	2707	1	0.01	2707	1	0.01
ulysses22	7013	7013	11,028,276	1,800.00	7013	2	0.01	7013	2	0.01	7013	1	0.09
gr24	1272	1272	969,837	193.34	1272	46	0.05	1272	6	0.01	1272	1	0.01
fr126	937	937	11,402,433	1,800.00	937	52	0.05	937	2	0.01	937	1	0.01
bay29	1610	1610	6,393,643	1,800.00	1610	54	0.07	1610	6	0.01	1610	1	0.00
bays29	2020	-	5,961,391	1,800.00	2020	64	0.08	2020	20	0.02	2020	1	0.03
dantzig42	699	-	3,939,003	1,800.00	699	92	0.22	699	11	0.02	699	1	0.01
swiss42	1273	-	3,788,982	1,800.00	1273	98	0.24	1273	6	0.02	1273	1	0.03
att48	10628	-	2,169,469	1,800.00	10628	132	0.53	10628	25	0.05	10628	3	0.12
gr48	5046	-	2,627,539	1,800.00	5046	18,994	47.15	5046	2,252	1.78	5046	1	0.07
hk48	11461	-	2,549,620	1,800.00	11461	94	0.42	11461	12	0.03	11461	1	0.03
eil51	426	-	2,128,857	1,800.00	426	2,116	4.91	426	361	0.29	426	1	0.04
berlin52	7542	-	2,493,694	1,800.00	7542	1	0.01	7542	1	0.01	7542	1	0.03
brazil158	25395	-	2,653,493	1,800.00	25395	788	2.54	25395	25	0.08	25395	1	0.08
st70	675	-	1,429,342	1,800.00	675	14,734	55.30	675	1,452	2.00	675	1	0.08
eil76	538	-	1,047,974	1,800.00	538	514	3.75	538	335	0.61	538	1	0.03
gr96	55209	-	1,052,177	1,800.00	55610	180,475	1,800.00	55209	317,005	1,015.46	55209	3	2.13
rat99	1211	-	926,581	1,800.01	1211	2,076	18.61	1211	742	2.02	1211	1	0.12
kroC100	20749	-	707,868	1,800.00	21332	208,259	1,800.00	20749	42,690	134.95	20749	1	0.13
kroD100	21294	-	553,556	1,800.00	21536	155,820	1,800.00	21294	3,382	13.26	21294	1	0.16
rd100	7910	-	562,115	1,800.00	7910	1,236	19.74	7910	27	0.22	7910	1	0.08
eil101	629	-	800,340	1,800.00	629	3,186	40.43	629	1,337	4.09	629	1	0.10
lin105	14379	-	1,222,012	1,800.00	14379	206	8.91	14379	12	0.20	14379	1	0.07
pr107	44303	-	3,561,595	1,800.00	44303	284	10.42	44303	45	1.45	44303	1	0.15
gr120	6942	-	415,518	1,800.00	-	66,083	1,800.00	6942	19,231	104.65	6942	1	0.26
pr124	59030	-	791,522	1,800.01	59649	57,621	1,800.00	59030	123,938	697.88	59030	1	0.31
pr144	58537	-	4,372,852	1,800.00	-	58,007	1,800.00	58537	3,237	34.03	58537	1	0.40
pr152	73682	-	1,727,783	1,800.02	-	79,708	1,800.00	73682	33,602	382.80	73682	1	0.84
ul159	42080	-	402,534	1,800.01	42309	93,458	1,800.00	42080	188,943	1,689.91	42080	1	0.14
pr226	80369	-	194,824	1,800.01	-	21,402	1,800.00	80369	5,980	97.01	80369	1	0.40
pr264	49135	-	61,557	1,800.01	49273	6,054	1,800.00	49135	994	36.66	49135	1	0.47

Table 3 Experimental results on symmetric instances from TSPLIB. The instances are ordered by non-decreasing size (indicated by the number in each instance name). For each instance, we report the upper bound (UB) computed with the Lin-Kernighan-Helsgaun heuristic. For each method, we report the best solution found, the number of search tree nodes, and the solving time in seconds. The solving time limit was set to 1,800 seconds.

size	1-tree with filtering			AP with filtering			1-tree + AP with filtering		
	solved	time	nodes/s	solved	time	nodes/s	solved	time	nodes/s
50	1.00	0.17	144.77	1.00	0.15	3,581.80	1.00	0.17	147.66
100	1.00	1.75	41.43	1.00	10.93	1,247.26	1.00	1.44	39.68
150	1.00	21.73	18.47	0.97	82.20	622.48	1.00	15.84	16.54
200	1.00	242.75	13.67	1.00	14.08	493.02	1.00	105.40	12.26
250	0.44	1,351.33	11.75	1.00	87.79	329.74	0.71	921.04	10.56
300	0.37	1,268.25	7.14	0.97	168.73	237.42	0.60	975.86	6.42
350	0.13	1,571.26	5.78	0.93	174.69	200.83	0.30	1,360.32	5.27
400	0.13	1,594.88	4.50	0.77	545.65	154.61	0.23	1,475.17	4.16
450	0.13	1,674.32	3.63	0.87	344.65	151.10	0.20	1,505.84	3.38

Table 4 Comparing the 1-tree relaxation (1-tree), the assignment problem relaxation (AP), and the additive bounding relaxation (1-tree + AP), on randomly generated asymmetric TSP instances. For each method and size class, we report the fraction of instances solved, average solving time, and average number of search nodes processed per second. All reported times are in seconds; the solving time limit was set to 1,800 seconds.

instance	UB	IBM ILOG CP Optimizer			1-tree no filtering			1-tree with filtering		
		best tour	search nodes	time	best tour	search nodes	time	best tour	search nodes	time
br17	39	39	18,862,474	1,202.83	39	344,170	66.35	39	223,603	34.40
ftv33	1286	1286	1,239,425	589.51	1286	65	0.28	1286	3	0.03
ftv35	1473	1473	3,669,666	1,800.00	1473	131	0.77	1473	41	0.12
ftv38	1530	-	3,337,894	1,800.00	1530	323	1.62	1530	87	0.22
ftv44	1613	-	2,730,136	1,800.00	1613	717	6.11	1613	227	0.72
ftv47	1776	-	2,199,876	1,800.00	1776	1,495	9.04	1776	471	1.36
ry48p	14422	-	1,531,267	1,800.00	14422	1,115	13.43	14422	364	1.58
ft53	6905	-	4,800,410	1,800.00	6905	1	0.16	6905	1	0.10
ftv55	1608	-	1,543,447	1,800.00	1608	5,021	41.77	1608	2,155	6.74
ftv64	1839	-	904,623	1,800.00	1839	10,413	138.15	1839	2,111	11.56
ft70	38673	-	699,906	1,800.00	38673	277	9.28	38673	138	1.26
ftv70	1950	-	918,180	1,800.00	1950	32,199	624.26	1950	5,992	42.01
kro124p	36230	-	332,249	1,800.00	36230	20,617	1,257.89	36230	5,670	91.77

Table 5 Experimental results on asymmetric instances from TSPLIB. The instances are ordered by non-decreasing size (indicated by the number in each instance name). For each instance, we report the upper bound (UB) computed with the Lin-Kernighan-Helsgaun heuristic. For each method, we report the best solution found, the number of search tree nodes, and the solving time in seconds. The solving time limit was set to 1,800 seconds.

instance	UB	1-tree with filtering			AP with filtering			1-tree + AP with filtering		
		best tour	search nodes	time	best tour	search nodes	time	best tour	search nodes	time
br17	39	39	223,603	34.40	39	5,716,951	199.29	39	221,738	35.15
ftv33	1286	1286	3	0.03	1286	33,936	5.71	1286	3	0.03
ftv35	1473	1473	41	0.12	1473	14,613	2.37	1473	88	0.20
ftv38	1530	1530	87	0.22	1530	48,260	9.83	1530	102	0.27
ftv44	1613	1613	227	0.72	1613	63,362	14.48	1613	169	0.56
ftv47	1776	1776	471	1.36	1776	9,094,310	1,800.00	1776	331	1.07
ry48p	14422	14422	364	1.58	-	3,905,158	1,800.00	14422	155	0.76
ft53	6905	6905	1	0.10	-	7,225,825	1,800.00	6905	1	0.08
ftv55	1608	1608	2,155	6.74	-	5,502,314	1,800.00	1608	2,363	7.39
ftv64	1839	1839	2,111	11.56	1878	4,590,304	1,800.00	1839	1,896	10.45
ft70	38673	38673	138	1.26	-	3,761,251	1,800.00	38673	86	0.91
ftv70	1950	1950	5,992	42.01	-	3,798,786	1,800.00	1950	2,983	21.97
kro124p	36230	36230	5,670	91.77	-	1,418,885	1,800.00	36230	5,907	91.60

Table 6 Comparing propagation based on the 1-tree relaxation (1-tree), the Assignment Problem relaxation (AP), and the additive bounding relaxation (1-tree + AP), on asymmetric instances from TSPLIB. For each instance, we report the upper bound (UB) computed with the Lin-Kernighan-Helsgaun heuristic. For each solving method, we report the best solution found, the number of search tree nodes, and the solving time in seconds. The solving time limit was set to 1,800 seconds.

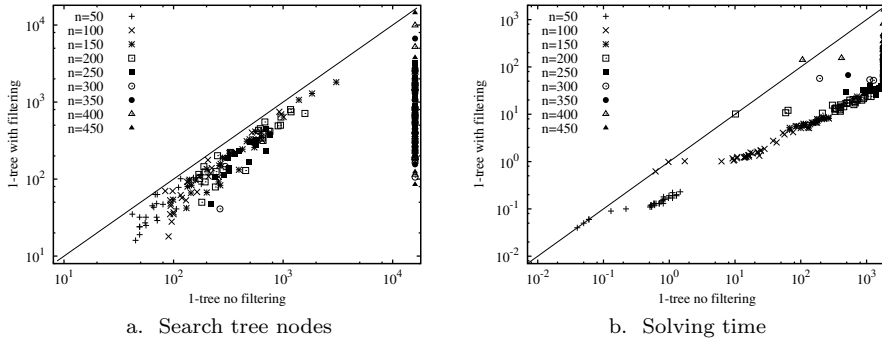


Fig. 5 Comparing the 1-tree relaxation with and without filtering on randomly generated asymmetric TSP instances, in terms of number of search tree nodes (a) and solving time in seconds (b). The solving time limit was set to 1,800 seconds. Each data point in these figures corresponds to an instance.

performs very well on problem instances of this type, which confirms the findings of Focacci et al. [20]. We also observe that the additive bounding procedure can improve the 1-tree relaxation, although the improvements are not dramatic.

We next consider asymmetric problem instances from TSPLIB. We first compare the 1-tree relaxation, with and without filtering, to IBM ILOG CP Optimizer in Table 5, which follows the same format as Table 3. Again, we observe that adding only the bound from the 1-tree relaxation already greatly improves the standard CP model, while our filtering methods (1-tree with filtering) make it possible to solve these problems more efficiently.

Lastly, we compare the three different relaxations (1-tree, assignment problem, and the additive bound) on these instances. The results are reported in Table 6, which again follows the same format as Table 3. Somewhat to our surprise, the assignment problem relaxation (AP with filtering) did not perform as well as the 1-tree relaxation on these instances (1-tree with filtering). However, when the assignment problem is coupled with the 1-tree relaxation to produce the additive bound, several instances can be solved much faster. In particular, with the additive bounding relaxation the solving time for instance *ftv70* is almost halved with respect to the 1-tree relaxation.

8 Conclusion

We have presented new propagation methods for handling the **weighted-circuit** constraint based on the 1-tree relaxation. We proposed to define the **weighted-circuit** constraint using a set variable that represents the mandatory and possible edges that can be taken in a solution. This representation naturally allows to apply and combine different relaxations, and we have in particular described an additive bounding procedure that combines the 1-tree relaxation with the assignment problem relaxation in the context of domain filtering for the **weighted-circuit** constraint.

We have implemented our filtering algorithms and evaluated their performance on Traveling Salesman Problems (TSPs). Our experimental results have demonstrated that domain filtering based on the 1-tree relaxation can indeed be very effective, and outperforms existing CP technology on the considered problem set. In particular, our

approach allows to scale up the best known CP-based technology from small-sized (up to around 50 cities) to medium-sized problems instances (up to around 150 cities) for symmetric TSPs.

References

1. E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn. SCIL – Symbolic Constraints in Integer Linear Programming. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 75–87. Springer, 2002.
2. D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.
3. F. Azevedo. Cardinal: A Finite Sets Constraint Solver. *Constraints*, 12:93–129, 2007.
4. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.
5. N. Beldiceanu, P. Flener, and X. Lorca. The Tree Constraint. In *Proceedings of the Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 3524 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2005.
6. C. Bessiere. Constraint Propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.
7. G. Carpaneto, S. Martello, and P. Toth. Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13(1):191–223, 1988.
8. G. Carpaneto, M. Dell’Amico, and P. Toth. Exact Solution of Large-Scale, Asymmetric Traveling Salesman Problems. *ACM Transactions on Mathematical Software*, 21(4):394–409, 1995.
9. Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *Proceedings of the 14th International Conference on Logic Programming (ICLP)*, pages 316–330. MIT Press, 1997.
10. B. Chazelle. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.
11. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
12. B. Dixon, M. Rauch, and R. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.
13. G. Dooms and I. Katriel. The *minimum spanning tree* constraint. In *Proceedings of CP*, volume 4204 of *LNCS*, pages 152–166. Springer, 2006.
14. G. Dooms and I. Katriel. The “not-too-heavy spanning tree” constraint. In *Proceedings of the Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 4510 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2007.
15. J.-G. Fages and X. Lorca. Revisiting the Tree Constraint. In *Proceedings of the 17th International Conference on the Principles and Practice of Constraint Programming (CP)*, volume 6876 of *LNCS*, pages 271–285. Springer, 2011.
16. M. Fischetti and P. Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37(2):319–328, 1989.

17. M. Fischetti and P. Toth. An additive bounding procedure for the asymmetric travelling salesman problem. *Mathematical Programming*, 53(1):173–197, 1992.
18. F. Focacci. *Solving Combinatorial Optimization Problems in Constraint Programming*. PhD thesis, University of Ferrara, 2001.
19. F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203, 1999.
20. F. Focacci, A. Lodi, M. Milano, and D. Vigo. Solving TSP through the integration of OR and CP techniques. *Electronic Notes in Discrete Mathematics*, 1:13–25, 1999.
21. F. Focacci, A. Lodi, and M. Milano. Embedding relaxations in global constraints for solving TSP and TSPTW. *Annals of Mathematics and Artificial Intelligence*, 34(4):291–311, 2002.
22. F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002.
23. L. Genç Kaya and J.N. Hooker. A filter for the circuit constraint. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 4204 of *Lecture Notes in Computer Science*, pages 706–710. Springer, 2006.
24. C. Gervet. New structures of symbolic constraint objects: sets and graphs. In *Third Workshop on Constraint Logic Programming (WCLP'2003)*, 1993.
25. C. Gervet. Constraints over Structured Domains. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 17. Elsevier, 2006.
26. M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.
27. G. Gutin and A.P. Punnen, editors. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.
28. M. Held and R.M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18:1138–1162, 1970.
29. M. Held and R.M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees: Part II. *Mathematical Programming*, 1:6–25, 1971.
30. K. Helsgaun. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.
31. K. Helsgaun. General k-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation*, 1(2):119–163, 2009.
32. W.-J. van Hoes and I. Katriel. Global Constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 6. Elsevier, 2006.
33. IBM ILOG CP V1.6 User Manual, 2010.
34. IBM ILOG OPL V12.2 User Manual, 2010.
35. R. Jonker and T. Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161–163, 1983.
36. R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Animations*, pages 85–103. Plenum Press, 1972.
37. P. Kilby and P. Shaw. Vehicle Routing. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 23. Elsevier, 2006.
38. H.W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

-
39. J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.
 40. A. Lodi, M. Milano, and L.-M. Rousseau. Discrepancy-Based Additive Bounding Procedures. *INFORMS Journal on Computing*, 18(4):480–493, 2006.
 41. M. Milano and W.J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.
 42. G. Pesant, M. Gendreau, J.Y. Potvin, and J.M. Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.
 43. R.C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.
 44. J.F. Puget. PECOS: a high level constraint programming language. In *Proceedings of the Singapore International Conference on Intelligent Systems (SPICIS)*, 1992.
 45. J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 362–367. AAAI Press, 1994.
 46. J.-C. Régin. Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint. In *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 5015 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2008.
 47. J.-C. Régin. Global Constraints: a survey. In P. Van Hentenryck and M. Milano, editors, *Hybrid Optimization*, pages 63–134. Springer, 2011.
 48. J.-C. Régin, L.-M. Rousseau, M. Rueher, and W.-J. van Hoeve. The Weighted Spanning Tree Constraint Revisited. In *Proceedings of the Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 6140 of *LNCS*, pages 176–180. Springer, 2010.
 49. M. Sellmann. Theoretical Foundations of CP-based Lagrangian Relaxation. In *Proceedings of the 10th International Conference on the Principles and Practice of Constraint Programming (CP)*, volume 3258 of *LNCS*, pages 634–647. Springer, 2004.
 50. R. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters*, 14(1):30–33, 1982.
 51. R.E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.