Journal of Combinatorial Optimization (2024) 47:4
https://doi.org/10.1007/s10878-023-01097-4

A branch-and-cut algorithm for the balanced traveling salesman problem

Thi Quynh Trang Vo[1] · **Mourad Baiou**[1] · **Viet Hung Nguyen**[1]

Accepted: 28 November 2023 / Published online: 28 January 2024

Abstract
The balanced traveling salesman problem (BTSP) is a variant of the traveling salesman problem, in which one seeks a tour that minimizes the difference between the largest and smallest edge costs in the tour. The BTSP, which is obviously NP-hard, was first investigated by Larusic and Punnen (Comput Oper Res 38(5):868–875, 2011). They proposed several heuristics based on the double-threshold framework, which converge to good-quality solutions though not always optimal. In this paper, we design a special-purpose branch-and-cut algorithm for exactly solving the BTSP. In contrast with the classical TSP, due to the BTSP's objective function, the efficiency of algorithms for solving the BTSP depends heavily on determining correctly the largest and smallest edge costs in the tour. In the proposed branch-and-cut algorithm, we develop several mechanisms based on local cutting planes, edge elimination, and variable fixing to locate those edge costs more precisely. Other critical ingredients in our method are algorithms for initializing lower and upper bounds on the optimal value of the BTSP, which serve as warm starts for the branch-and-cut algorithm. Experiments on the same testbed of TSPLIB instances show that our algorithm can solve 63 out of 65 instances to proven optimality.

Keywords Traveling salesman problem · Balanced optimization · Mixed-integer programming · Branch-and-cut

Mathematics Subject Classification 90-10 · 90-05

✉ Viet Hung Nguyen
viet_hung.nguyen@uca.fr

Thi Quynh Trang Vo
thi_quynh_trang.vo@uca.fr

Mourad Baiou
mourad.baiou@uca.fr

[1] INP Clermont Auvergne, Univ Clermont Auvergne, Mines Saint-Etienne, CNRS, UMR 6158 LIMOS, 1 Rue de la Chebarde, Aubiere Cedex, France

1 Introduction

Given a finite set E with cost vector $\boldsymbol{c}$ and a family $\mathcal{F}$ of feasible subsets of E, a balanced optimization problem seeks a feasible subset $S^* \in \mathcal{F}$ that minimizes the difference in cost between the most expensive and least expensive elements used, i.e., $\max_{e \in S^*} c_e - \min_{e \in S^*} c_e$. This optimization class arises naturally in many practical situations where one desires a fair distribution of costs. Balanced optimization was introduced by Martello (1984) in the context of the assignment problem. Then, a line of works was investigated for other specific cases of balanced optimization, such as the balanced shortest path (Turner 2011; Cappanera and Scutella 2005), the balanced minimum cut (Katoh and Iwano 1994), the balanced spanning tree (Galil and Schieber 1988; Camerini et al. 1986) and the balanced network flow (Scutellà 1998).

In this paper, we consider the balanced version of the traveling salesman problem (TSP). In the TSP's context, the finite set E is the edge set of a graph, and the feasible subset family $\mathcal{F}$ is the set of all Hamiltonian cycles (a.k.a tours) in the graph. The BTSP finds a tour whose difference between the largest and smallest edge costs is minimum. For abbreviation, we call this difference *the max-min distance*. Formally, given an undirected graph $G = (V, E)$ and a cost vector $\boldsymbol{c}$ associated with E, the BTSP can be formulated as follows:

$$\min_{\mathcal{H} \in \Pi(G)} \{\max_{e \in \mathcal{H}} c_e - \min_{e \in \mathcal{H}} c_e\} \tag{1}$$

where $\Pi(G)$ is the set of all Hamiltonian cycles in G. The BTSP is NP-hard by folklore that the problem of finding a Hamiltonian cycle in the graph can be reduced to a BTSP with unit edge costs.

The BTSP was first studied by Larusic and Punnen (2011) with applications in many practical problems, such as the nozzle guide vane assembly problem (Plante et al. 1987) and the cyclic workforce scheduling problem (Vairaktarakis 2003). While most of the previous works about balanced optimization focused on polynomial-time algorithms, the BTSP was the first NP-hard case studied.

Observe that the BTSP can be reduced to the problem of finding the shortest interval such that all edges whose costs are in the interval can form a Hamiltonian cycle. An approach for finding such an interval is the double-threshold algorithm (Martello 1984), widely used for balanced optimization problems. As its name suggests, the double-threshold algorithm maintains two thresholds of the tour's edge costs: a lower threshold and an upper threshold. At each iteration, the algorithm generates a threshold pair and checks whether the graph whose edge costs are restricted by this threshold pair is Hamiltonian. The interval to find is a threshold pair with the smallest difference.

A critical issue of this approach is that it requires solving $O(|E|)$ Hamiltonicity verification problems, which are NP-hard. It causes the approach to be unpractical when the problem size is large. To tackle this issue, Larusic and Punnen (2011) heuristically solved the Hamiltonicity verification problem at every iteration. They also developed four variants of the double-threshold algorithm to reduce the number of iterations without sacrificing solution quality by using the bottleneck TSP (LaRusic and Punnen 2014) and the maximum scatter TSP (Arkin et al. 1999). With these modifications, their

algorithms produced good-quality solutions within 10% optimality, estimated based on lower bounds, to 65 TSPLIB instances (Reinelt 1991). Furthermore, 27 solutions were provably optimal.

To the best of our knowledge, no exact algorithm based on Mixed-Integer Linear Programming (MILP) for the BTSP has been proposed in the literature, although it is pretty easy to formulate the BTSP through the existing MILP formulations for the TSP. The reason is that solving the BTSP's formulations directly without tools to locate the largest and smallest edge costs can be inefficient and more difficult than solving the classical TSP. In this paper, we propose a branch-and-cut algorithm that includes mechanisms to tighten the bounds of the largest and smallest edge costs. These mechanisms include local cutting planes, edge eliminating, and variable fixing techniques. To further improve the performance, we develop algorithms to initialize lower and upper bounds on the BTSP's optimal value. The efficiency of the proposed branch-and-cut algorithm is assessed through computational comparison to the double-threshold-based algorithms in Larusic and Punnen (2011). Numerical results show that our algorithm can solve to proven optimality 63 instances out of 65 within 3 h of CPU time limit.

The paper is organized as follows. Section 2 presents a MILP formulation for the BTSP. In Sect. 3, we propose a branch-and-cut algorithm for the BTSP with detailed descriptions of additional components: a lower bounding algorithm (Sect. 3.1), a local search algorithm (Sect. 3.2), edge elimination (Sect. 3.3), a family of local cutting planes called *local bounding cuts* (Sect. 3.4), variable fixing (Sect. 3.5), and separation strategies (Sect. 3.6). The algorithm's effectiveness is evaluated through computational results in Sect. 4. Finally, we give some conclusions in Sect. 5.

1.1 Preliminaries

Given a graph $G = (V, E)$ and a cost vector $\boldsymbol{c}$ associated with E, we provide below some notations used throughout the paper. For any subset S of V, let $\delta(S)$ be a subset of E where each edge has exactly one end-vertex in S, i.e., $\delta(S) = \{(i, j) \in E \mid i \in S \text{ and } j \in V \backslash S\}$. For abbreviation, we write $\delta(v)$ instead of $\delta(\{v\})$ for all $v \in V$. Given a Hamiltonian cycle $\mathcal{H} \in \Pi(G)$, we respectively denote by $u_{\mathcal{H}}$ and $l_{\mathcal{H}}$ the largest and smallest edge costs in $\mathcal{H}$. For an edge set $F \subseteq E$, we denote $V(F)$ the end-vertices set of edges in F and $C(F) = \{c_e \in \boldsymbol{c} \mid e \in F\}$ the edge cost set corresponding to F. Without loss of generality, we assume that $C(E) = \{C_1, \ldots, C_p\}$ where $p \leq m$ is the number of distinct components of the cost vector $\boldsymbol{c}$ and $C_1 < C_2 < \cdots < C_p$. For an interval $[\alpha, \beta]$, $G[\alpha, \beta]$ stands for a subgraph of G with edge set $E[\alpha, \beta] = \{e \in E \mid \alpha \leq c_e \leq \beta\}$. We call $G[\alpha, \beta]$ *the subgraph restricted by* $[\alpha, \beta]$. For any positive integer n, let $[n] = \{1, \ldots, n\}$.

2 MILP formulation for the BTSP

Given an undirected graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, the BTSP consists in finding a tour that minimizes the max-min distance. We denote by $\{x_e \mid e \in E\}$ a set of binary

variables where $x_e = 1$ if edge e is in the tour and $x_e = 0$ otherwise. Let u and l respectively be variables representing the tour's highest and smallest edge costs. We propose a MILP formulation for the BTSP as follows:

$$
\begin{aligned}
\text{(MILP-BTSP)} \quad \min \quad & u - l && && \text{(2a)}\\
\text{s.t.} \quad & \sum_{e \in \delta(v)} x_e = 2 && \forall v \in V && \text{(2b)}\\
& \sum_{e \in \delta(S)} x_e \geq 2 && \forall \emptyset \neq S \subset V && \text{(2c)}\\
& u \geq c_e x_e && \forall e \in E && \text{(2d)}\\
& l \leq c_e x_e + (1 - x_e) M_e && \forall e \in E && \text{(2e)}\\
& x_e \in \{0, 1\} && \forall e \in E && \text{(2f)}
\end{aligned}
$$

where $M_e = \min\{\max_{e' \in \delta(i)} c_{e'}, \max_{e' \in \delta(j)} c_{e'}\}$ for all $e = (i, j) \in E$. The objective function (2a) corresponds to the max-min distance. Constraints (2b) are degree constraints, which ensure that each vertex has precisely two incident edges in the tour. Constraints (2c) are the well-known subtour elimination inequalities that prevent the existence of subtours. Constraints (2d) and (2e) are used to estimate the highest and smallest edge costs. More specifically, constraints (2d) ensure that u must be greater than or equal to the costs of edges selected in the tour. On the other hand, if an edge e occurs in the tour (i.e., $x_e = 1$), inequalities (2e) read as $l \leq c_e$, which are true by the definition of l. Otherwise (i.e., $x_e = 0$), constraints (2e) become $l \leq M_e$, which are valid as $l \leq \max_{e \in \delta(i)} c_e, \forall i \in V$.

Obviously, (MILP-BTSP) can be solved by branch-and-cut algorithms for the TSP where subtour elimination constraints 2c are generated as cutting planes. However, the BTSP's main challenge is not representing tours but estimating the largest and smallest edge costs. The absence of mechanisms to address this task causes general-purpose branch-and-cut algorithms can not solve the BTSP efficiently.

This issue can be demonstrated by the following experiment. We solve the TSP and BTSP on the TSPLIB instance si175 (with 175 vertices) (Reinelt 1991) by the commercial solver CPLEX 12.10, a widely-used general-purpose branch-and-cut algorithm for MILP problems. We formulate both problems by the same tour's constraints, i.e., degree constraints (2b), subtour elimination constraints (2c), and integral constraints (2f). The CPU time limit is set to 10,800 s. Table 1 shows the results of the two problems. Column "*CPU(s)*" indicates the CPU runtime in seconds of the solver. Column "*Gap(%)*" gives the current best IP relative gap (i.e., $|\text{LB} - \text{UB}|/\text{UB}$ where LB, UB are respectively the lower and upper bounds). Columns "*Nodes*" and "*Depth*" respectively report the number of nodes and the depth of the enumeration tree (i.e., the number of edges along the longest path from the root node to a leaf node).

As shown in Table 1, while the TSP is solved quickly in 25 s, the BTSP can not be solved to proven optimality within the CPU time limit, i.e., the best IP relative gap value is 28.57%. The size and depth of the enumeration tree of the BTSP are also substantially larger than those of the TSP. Thus, designing a specific-purpose branch-and-cut algorithm for the BTSP is crucial.

Table 1 The results of the TSP and BTSP on the instance si175 solved by CPLEX 12.10

Problem	CPU (s)	Gap (%)	Nodes	Depth
TSP	25.52	0.00	4324	89
BTSP	10,800.09	28.57	140,003	987

3 Branch-and-cut algorithm

In this section, we propose a dedicated branch-and-cut algorithm for exactly solving the BTSP. Our branch-and-cut algorithm contains mechanisms to locate the largest and smallest edge costs (i.e., local bounding cuts, edge elimination, and variable fixing) and algorithms to initialize lower and upper bounds on the optimal value of the BTSP. We first provide the algorithm's general schema, followed by an in-depth description of its vital components.

Figure 1 illustrates the flowchart of our branch-and-cut algorithm for the BTSP. After the initialization step, the algorithm constructs an enumeration tree where each node corresponds to a linear programming (LP) relaxation of (MILP-BTSP). The tree's root node is associated with an LP relaxation obtained by dropping all subtour elimination constraints (2c) and the integrality requirements on the variables $x_e, e \in E$. At each iteration, we select an active node (that has yet to be pruned or branched on) and solve the corresponding LP relaxation. If we obtain an optimal solution in which all x_e values are integral, we verify the satisfaction of subtour elimination constraints and add violated ones to the LP relaxations. Otherwise, we can solve separation problems to generate cutting planes in order to strengthen LP relaxations or select a variable having a fractional value to branch.

Our improvements focus on initializing, enhancing the upper bound, and generating cutting planes on the enumeration tree. Other components of the branch-and-cut algorithm, such as node selection or branching, follow the default rules of the commercial solver CPLEX 12.10.

The tightness of lower and upper bounds on the optimal value of the BTSP is crucial for pruning the enumeration tree. To initialize good bounds, we develop a lower bounding algorithm based on the biconnectivity of Hamiltonian cycles (Sect. 3.1) and a local search algorithm for the BTSP based on k-opt algorithms (Lin 1965; Helsgaun 2009) (Sect. 3.2). The bounds found by these algorithms are used not only to provide warm starts for the branch-and-cut algorithm but also to sparsify the graph by eliminating edges that can not be part of an optimal solution (Sect. 3.3).

During the branch-and-cut algorithm, to further improve the upper bound, when a new incumbent solution is found, we perform our local search algorithm for the BTSP to enhance this solution. To tighten the lower bound, we develop several valid inequalities for the BTSP, i.e., *local bounding cuts* (Sect. 3.4) and *variable fixing cuts* (Sect. 3.5). Moreover, we also propose strategies to decide whether to generate cuts or to branch at a node of the tree, which are presented in Sect. 3.6.

In the remainder of this section, we present these components in detail.

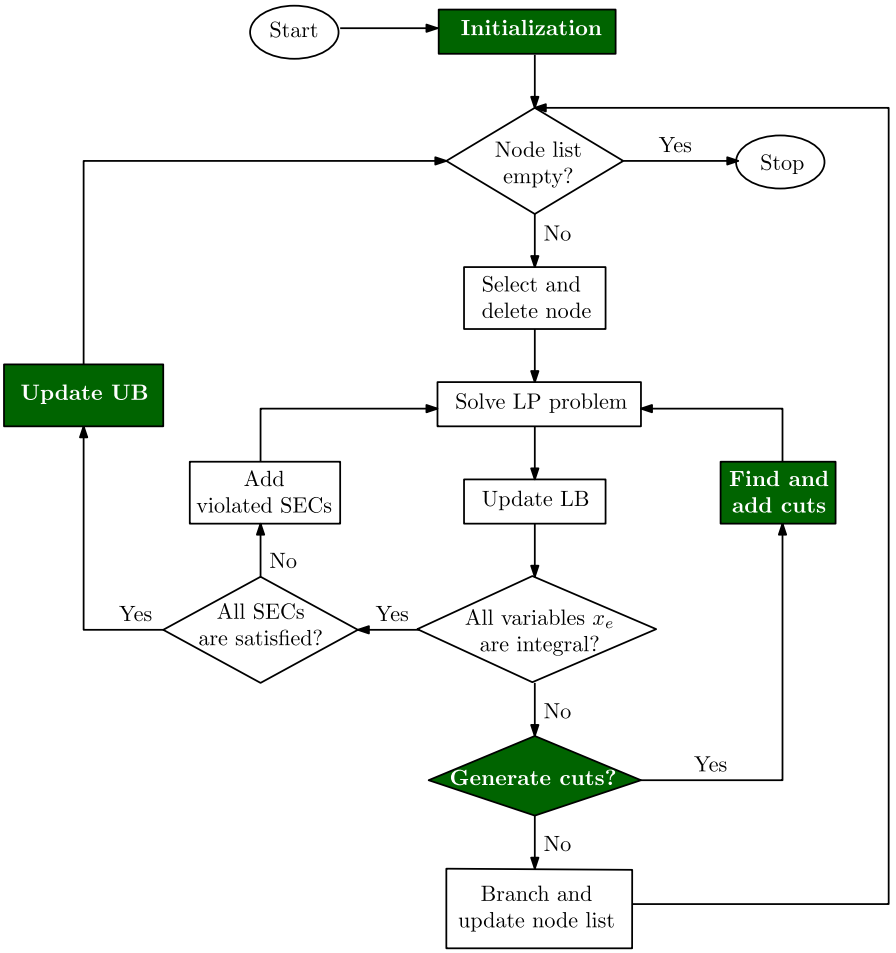

Fig. 1 The schema of our branch-and-cut algorithm for the BTSP. Our contributions focus on the green components in the diagram

3.1 Lower bounding algorithm

Given a graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, we present below an algorithm partly inspired by the Hamiltonian verification procedure in Larusic and Punnen (2011) to compute a lower bound on the optimal value of the BTSP at the initialization step.

As mentioned in Larusic and Punnen (2011), a Hamiltonian graph must be a biconnected graph (i.e., a graph in which for any pair of vertices u and v, there exist two paths from u to v without any vertices in common except u and v). The intuition of the algorithm is that for all distinct costs $C_i \in C(E)$, we find the shortest interval containing C_i such that the subgraph restricted by this interval is biconnected. The minimum length among these intervals is a lower bound on the minimum max-min distance of the BTSP. Algorithm 1 gives a formal description of our lower bounding

algorithm. Before describing the algorithm in detail, we introduce some definitions and lemmas.

Algorithm 1 Lower bounding algorithm for the BTSP

Require: A graph $G = (V, E)$ with edge costs $\boldsymbol{c}$.
Ensure: A lower bound on the optimal value of the BTSP.
Let $C_1 < C_2 < \cdots < C_p$ be the distinct costs of $\boldsymbol{c}$
$b_0 \leftarrow 1; b_i \leftarrow p+1, \forall i \in [p]; C_{p+1} \leftarrow +\infty$
for $i \in [p]$ **do**
  $j \leftarrow b_{i-1}$
  while $j \leq p$ **do**
    if $G[C_i, C_j]$ is biconnected **then**
      $b_i \leftarrow j$
      break
    end if
    $j \leftarrow j+1$
  end while
end for
for $i \in [p]$ **do**
  $l_i \leftarrow 1, u_i \leftarrow p$
  for $j \in [i]$ **do**
    if $C_{b_j} - C_j < C_{u_i} - C_{l_i}$ **then**
      $l_i \leftarrow j, u_i \leftarrow b_j$
    end if
  end for
end for
return $\min_{i \in [p]} C_{u_i} - C_{l_i}$.

Definition 1 (*Biconnected interval*) For any $C_i \in C(E)$, a *biconnected interval compatible with* C_i is an interval $[\alpha, \beta]$ such that

(i) $\alpha \leq C_i \leq \beta$;
(ii) $G[\alpha, \beta]$ is biconnected.

The length of a biconnected interval $[\alpha, \beta]$ is the difference between β and α, i.e., $\beta - \alpha$. We denote by $\gamma(C_i)$ the length of the shortest biconnected interval compatible with C_i.

Lemma 1 *Let $\mathcal{H}$ be a tour in G. If $\mathcal{H}$ contains an edge with cost C_i, then*

$$u_{\mathcal{H}} - l_{\mathcal{H}} \geq \gamma(C_i).$$

Proof We consider the graph $G[l_{\mathcal{H}}, u_{\mathcal{H}}]$ with edge set $E[l_{\mathcal{H}}, u_{\mathcal{H}}] = \{e \in E \mid l_{\mathcal{H}} \leq c_e \leq u_{\mathcal{H}}\}$. $G[l_{\mathcal{H}}, u_{\mathcal{H}}]$ is biconnected as it contains the tour $\mathcal{H}$. Since $\mathcal{H}$ has an edge with cost C_i, $l_{\mathcal{H}} \leq C_i \leq u_{\mathcal{H}}$. Thus, $(l_{\mathcal{H}}, u_{\mathcal{H}})$ is a biconnected interval compatible with C_i. By the definition of $\gamma(C_i)$, $u_{\mathcal{H}} - l_{\mathcal{H}} \geq \gamma(C_i)$. □

Corollary 1 *Let $\gamma^* = \min_{C_i \in C(E)} \gamma(C_i)$ and $\mathcal{OPT}$ be the optimal value of (MILP-BTSP), we have $\gamma^* \leq \mathcal{OPT}$.*

Thanks to Corollary 1, to obtain a lower bound on the optimal value of the BTSP, it is sufficient to find the shortest biconnected interval compatible with C_i for all $C_i \in C(E)$. The following lemma provides a characterization of the shortest biconnected intervals.

Lemma 2 *If $[\alpha, \beta]$ is the shortest biconnected interval compatible with C_i, then α and β belong to the edge cost set of E.*

Proof We consider the graph $G[\alpha, \beta]$. Let $\alpha' = \min\{c_e \mid e \in E[\alpha, \beta]\}$ and $\beta' = \max\{c_e \mid e \in E[\alpha, \beta]\}$. Obviously, $\alpha', \beta' \in C(E)$ and $\alpha' \leq C_i \leq \beta'$. Since $G[\alpha', \beta'] = G[\alpha, \beta]$ and $G[\alpha, \beta]$ is biconnected, $G[\alpha', \beta']$ is also biconnected. Thus, $[\alpha', \beta']$ is a biconnected interval compatible with C_i.

Since $[\alpha, \beta]$ is the shortest biconnected interval compatible with C_i, $\beta - \alpha \leq \beta' - \alpha'$. On the other hand, by the definition of $G[\alpha, \beta]$, $\alpha \leq \alpha'$ and $\beta \geq \beta'$. Then, $\beta' - \alpha' \leq \beta - \alpha$. The equality holds if and only if $\alpha = \alpha'$ and $\beta = \beta'$. □

By Lemma 2, to find the shortest biconnected intervals, we first determine the smallest index $b_j \in [p]$ such that $G[C_j, C_{b_j}]$ is biconnected, for all $C_j \in C(E)$. Note that the b_j computation is according to the increasing ordering of j (i.e., starting with C_1, followed by C_2, and so on). Then, the shortest biconnected interval compatible with C_i is the shortest interval $[C_j, C_{b_j}]$ containing C_i. A naive way to find b_j is to initially set b_j by j and increase b_j until $G[C_j, C_{b_j}]$ is biconnected. It requires checking the graph's biconnectivity $O(|E|^2)$ times. We can reduce it to $O(|E|)$ by using the following lemma.

Lemma 3 *For any $i, j \in [p]$, if $C_i < C_j$ then $b_i \leq b_j$.*

Proof We prove the lemma by contradiction. Assume that there exist two costs C_i, C_j such that $C_i < C_j$ and $b_i > b_j$. Obviously, $G[C_j, C_{b_j}]$ is a subgraph of $G[C_i, C_{b_j}]$. Since $G[C_j, C_{b_j}]$ is biconnected, $G[C_i, C_{b_j}]$ is also biconnected. On the other hand, b_i is the smallest value such that $G[C_i, C_{b_i}]$ is biconnected. Thus, $b_i \leq b_j$, contradicts the assumption. □

Using Lemma 3, we can set b_j initially as b_{j-1} instead of j. This reduces the number of biconnectivity checks at most $O(|E|)$. The algorithm then repeatedly verifies the biconnectivity of the graph $G[C_j, C_{b_j}]$ and increases b_j until $G[C_j, C_{b_j}]$ is a biconnected graph. Since a biconnected graph is a connected graph without articulation vertices, the graph's biconnectivity can be checked in $O(|V| + |E|)$ by Tarjan's algorithm (Tarjan 1972). In total, the complexity of Algorithm 1 is $O(|E|^2)$.

3.2 Local search algorithm

We now propose a local search algorithm for the BTSP, called *k-balanced*, based on *k*-opt algorithms for the TSP (Lin 1965; Helsgaun 2009). The algorithm takes a graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, a tour $\mathcal{H}$, and a constant k as input and returns an improved tour with a smaller max-min distance. The purpose of this algorithm is twofold: i) to provide a good upper bound at the initialization step, and ii) to enhance the incumbent solution in the course of the branch-and-cut algorithm.

The intuition of k-balanced is to repeatedly perform k-exchanges (k-opt moves) to improve the current tour. A k-exchange replaces k edges in the current tour with k edges in such a way that a tour with a smaller max-min distance is achieved. Algorithm 2 sketches a generic version of k-balanced. In the following, we describe in detail the algorithm.

Algorithm 2 Generic k-balanced

Require: A graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, a tour $\mathcal{H}$, and a constant k.
Ensure: A tour $\mathcal{H}'$ such that $u_{\mathcal{H}'} - l_{\mathcal{H}'} < u_{\mathcal{H}} - l_{\mathcal{H}}$.
1: $improved \leftarrow$ **True**
2: **while** $improved$ **do**
3: $improved \leftarrow$ **False**
4: Select (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.
5: $EC(F, l', u') \leftarrow \{(i, j) \in E \mid i, j \in V(F) \wedge (l' \leq c_{(i,j)} \leq u')\}$.
6: **if** exists a k-subset $\overline{F} \subset EC(F, l', u')$ such that $(\mathcal{H} \backslash F) \cup \overline{F}$ is a tour **then**
7: $\mathcal{H} \leftarrow (\mathcal{H} \backslash F) \cup \overline{F}$.
8: $improved \leftarrow$ **True**
9: **end if**
10: **end while**
11: **return** $(\mathcal{H}, l_{\mathcal{H}}, u_{\mathcal{H}})$.

Given a tour $\mathcal{H}$ of G, at each iteration, k-balanced constructs two edge sets, $F = \{f_1, \ldots, f_k\}$ and $\overline{F} = \{\overline{f}_1, \ldots, \overline{f}_k\}$, such that $\mathcal{H}' = (\mathcal{H} \backslash F) \cup \overline{F}$ is a new tour with a smaller max-min distance. We call the edges of F *out-edges* and the edges of $\overline{F}$ *in-edges*.

The max-min distance of $\mathcal{H}'$ is smaller than that of $\mathcal{H}$ if and only if all edge costs of $\mathcal{H}'$ belong to an interval shorter than $[l_{\mathcal{H}}, u_{\mathcal{H}}]$. Due to this fact, the out-edge set F must contain all edges with either the maximum edge cost or the minimum edge cost in $\mathcal{H}$ and the in-edge set $\overline{F}$ only comprises edges with costs belonging to a range $[l', u']$ such that $u' - l' < u_{\mathcal{H}} - l_{\mathcal{H}}$. In order to avoid searching all possible intervals $[l', u']$, we simply consider intervals $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.

We first describe a way to construct the in-edge set $\overline{F}$ given a triple (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$. Let $EC(F, l', u') = \{(i, j) \in E \mid i, j \in V(F) \wedge (l' \leq c_{(i,j)} \leq u')\}$ the set of edges whose end-vertices are in $V(F)$ with costs between l' and u'. By its definition, $EC(F, l', u')$ is precisely the set of edges that can be used to reconnect a tour from $\mathcal{H} \backslash F$, namely that $\overline{F} \subset EC(F, l', u')$. To construct $\overline{F}$, we solve the problem of completing a Hamiltonian cycle from $\mathcal{H} \backslash F$ with only edges in $EC(F, l', u')$. With k fixed, we can solve the same problem on G' - a compressed version of G with at most $2k$ vertices. The construction of $\overline{F}$ is thus cheap since it is independent of the size of G. Figure 2 illustrates this idea.

We now present rules to select (F, l', u'). We create three variants of k-balanced corresponding to three selection rules for (F, l', u'): *k-balanced min*, *k-balanced max*, and *k-balanced extreme*.

Algorithm 3 describes the selection rule of (F, l', u') for k-balanced min/max. In these variants, we select F in such a way as to maximize the cardinality of $EC(F, l', u')$. We call this rule *the maximum candidate cardinality principle* (MCCP). In particular, for k-balanced min, we set $(l', u') = (l_{\mathcal{H}} + 1, u_{\mathcal{H}})$ and initialize F by

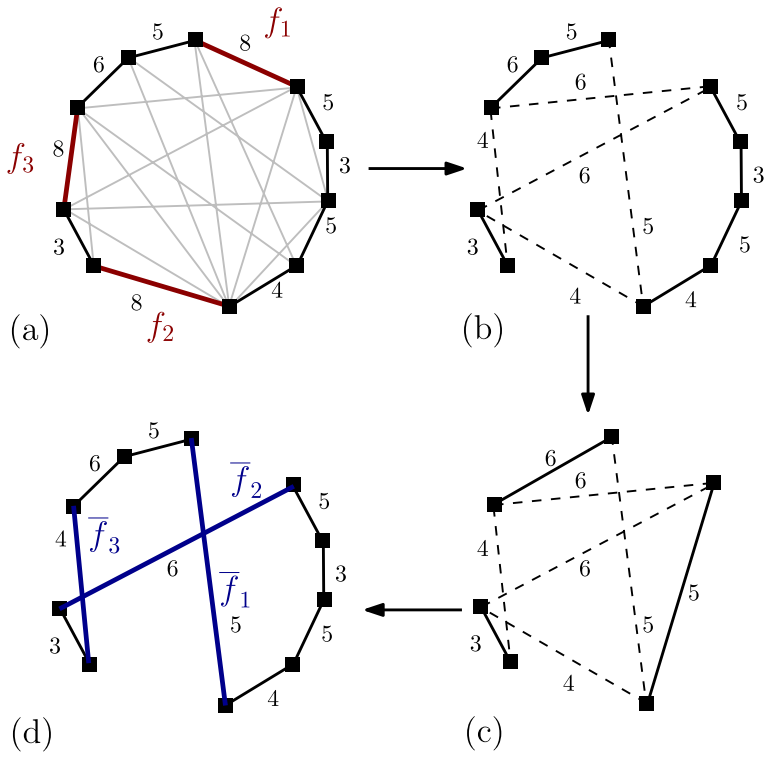

Fig. 2 Illustration of a 3-opt move in 3-balanced. (1.a) represents a tour $\mathcal{H}$ whose largest and smallest edge costs are 8 and 3, respectively. We will remove all edges with max-cost 8 (f_1, f_2, f_3) from $\mathcal{H}$ and set $(l', u') = (l_{\mathcal{H}}, u_{\mathcal{H}} - 1) = (3, 7)$. (1.b) illustrates the remainder $\mathcal{H}\backslash F$ of the tour. The dash lines are the edges of $EC(F, l', u')$ where edges have two endpoints in $V(F)$ and costs belong to $[3, 7]$. (1.c) demonstrates a compressed version G' of G, in which paths in $\mathcal{H}\backslash F$ are considered as edges. The problem of reconnecting $\mathcal{H}$ in G is equivalent to the one in G'. (1.d) shows the resulting tour with a smaller max-min distance, i.e. 3

all min-cost edges. At step i, an edge f_i in $\mathcal{H}\backslash F$ is added to the *current* F if it can increase the cardinality $EC(F, l', u')$ the most. More precisely, f_i is the edge that has the most incident edges having one end-vertex in $V(F)$ with costs between l' and u'. The selection procedure is repeated until the cardinality of F equals k. This selection rule is applied similarly for k-balanced max with two modifications: F initially is a set of all max-cost edges, and l', u' respectively equal $l_{\mathcal{H}}$ and $u_{\mathcal{H}} - 1$. Such a way to select (F, l', u') offers the uttermost cardinality of $EC(F, l', u')$ and thus increases the probability of $\overline{F}$'s existence. However, it slowly decreases the max-min distance at each iteration (the gain can be only 1 per k-exchange).

On the other hand, k-balanced extreme prioritizes dropping the max-min distance as fast as possible. While k-balanced min/max chooses edges to remove, the removal rule of k-balanced extreme is cost-based. For an edge cost $C_i \in C(E)$, let $d(C_i, \mathcal{H}) := \min(|l_{\mathcal{H}} - C_i|, |u_{\mathcal{H}} - C_i|)$. The out-edge set F is iteratively constructed as follows. Initially, $F = \emptyset$. At each iteration, we select a cost $C_i \in C(\mathcal{H}\backslash F)$ that

Algorithm 3 Selection rule for k-balanced min/max

Require: A graph $G = (V, E)$, a tour $\mathcal{H}$, a constant k, and an extreme type ET.
Ensure: (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.
1: **if** ET is min **then**
2: $F \leftarrow \{e \in \mathcal{H} \mid c_e = l_{\mathcal{H}}\}, l' \leftarrow l_{\mathcal{H}} + 1, u' \leftarrow u_{\mathcal{H}}$
3: **else if** ET is max **then**
4: $F \leftarrow \{e \in \mathcal{H} \mid c_e = u_{\mathcal{H}}\}, l' \leftarrow l_{\mathcal{H}}, u' \leftarrow u_{\mathcal{H}} - 1$
5: **end if**
6: **while** $|F| < k$ **do**
7: $f \leftarrow \arg\max_{e=(i,j)\in\mathcal{H}} \mid \delta(\{i, j\}) \cap \delta(F) \cap \{e' \in E | l' \le c_{e'} \le u'\} \mid$
8: $F \leftarrow F \cup \{f\}$
9: **end while**
10: **return** (F, l', u')

Algorithm 4 Selection rule for k-balanced extreme

Require: A graph $G = (V, E)$, a tour $\mathcal{H}$, and a constant k.
Ensure: (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.
1: $F \leftarrow \emptyset$.
2: **while** $|F| < k$ **do**
3: $removed\_cost \leftarrow \arg\min_{c_e \in C(\mathcal{H}\setminus F)} d(c_e, \mathcal{H})$
4: $F \leftarrow F \cup \{e \in \mathcal{H} \mid c_e = removed\_cost\}$
5: **end while**
6: $l' \leftarrow \min_{e\in\mathcal{H}\setminus F} c_e$
7: $u' \leftarrow \max_{e\in\mathcal{H}\setminus F} c_e$
8: **return** (F, l', u')

Table 2 Selection rules of (F, l', u')

	F	l'	u'
k-balanced min	min-cost edges and edges found by MCCP	$l_{\mathcal{H}} + 1$	$u_{\mathcal{H}}$
k-balanced max	max-cost edges and edges found by MCCP	$l_{\mathcal{H}}$	$u_{\mathcal{H}} - 1$
k-balanced extreme	extreme-cost edges and edges with smallest $d(c, \mathcal{H})$	$\min_{e\in\mathcal{H}\setminus F} c_e$	$\max_{e\in\mathcal{H}\setminus F} c_e$

yields the smallest $d(C_i, \mathcal{H})$ and add all the edges with cost C_i in $\mathcal{H}\setminus F$ to F. The process terminates when the cardinality of F is at least k. The tuple (l', u') is set to $(\min_{e\in\mathcal{H}\setminus F} c_e, \max_{e\in\mathcal{H}\setminus F} c_e)$. This selection method can reduce the max-min distance substantially. However, it also decreases the cardinality of $EC(F, l', u')$ and thus decreases the possibility of finding the in-edge set $\overline{F}$. Algorithm 4 gives the formal description of the rule.

Table 2 summarizes the (F, l', u') selection rules of the three k-balanced variants.

Notice that in all variants of k-balanced, we only consider one subset F to find k-exchange at each iteration. Although this setting can omit high-quality k-exchanges, it allows the algorithm to launch with many random initial tours and k's values within an acceptable amount of CPU time. Thus, we still can obtain reasonable feasible solutions. To further improve the algorithm, when the number of min-cost edges or max-cost edges is at most 3, we search 3-opt moves with all valid edge triples of the tour.

3.3 Edge elimination

In order to improve the efficiency of solving LP relaxations, it is helpful to eliminate edges that are not likely to be present in an optimal tour. It is important to note that the branch-and-cut algorithm aims to refine the incumbent solution. Consequently, if we can demonstrate that the occurrence of a specific edge would result in a worse tour compared to the current best tour, we can remove this edge to sparsify the graph and reduce the number of decision variables.

As proven in Lemma 1, if a tour contains an edge with cost C_i, its max-min distance is at least the length of the shortest biconnected interval compatible with C_i. Let $(\boldsymbol{x}_{\mathcal{H}^0}, l_{\mathcal{H}^0}, u_{\mathcal{H}^0})$ be the initial feasible solution corresponding to a tour $\mathcal{H}^0$ found by Algorithm 2. Then, edges with costs C_i satisfying $\gamma(C_i) > u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$ can not be part of an optimal tour; otherwise, the max-min distance of this tour will be greater than $u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$. By this observation, we can remove edges $e \in E$ such that $\gamma(c_e) > u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$.

3.4 Local bounding cuts

The BTSP entails estimating the largest and smallest edge costs compared to the TSP. This task is non-trivial and enormously impacts the algorithm's performance. In (MILP-BTSP), while the highest edge cost u is directly estimated through the edge variables, the smallest edge cost estimation needs to use the constants M_e. It can lead to untight bounds for l in LP relaxations and make solving the BTSP noticeably more time-consuming than solving the TSP. To tackle this issue, in this section, we provide a family of local cutting planes to strengthen bounds of the smallest edge cost in LP relaxations.

Observe that in the enumeration tree, each node is associated with an ordered pair $\langle F_0, F_1\rangle$ of two disjoint edge sets where F_0 and F_1 respectively contain edges whose corresponding variables have been fixed to 0 and 1. Given a node $\langle F_0, F_1\rangle$, a tour found by this node or its descendants is one whose incidence vector satisfies

$$\begin{aligned} x_e &= 0 \qquad \forall e \in F_0 \\ x_e &= 1 \qquad \forall e \in F_1. \end{aligned}$$

In other words, this tour permanently includes the edges of F_1 and excludes the edges of F_0. Let $I_{C(F_1)}$ be the minimum of $C(F_1) = \{c_e \mid e \in F_1\}$. Obviously, the smallest edge cost of the tour can not exceed $I_{C(F_1)}$. Based on this observation, we have the following inequalities, called *local bounding cuts*

$$l \le c_e x_e + (1 - x_e) I_{C(F_1)} \quad \forall e \in E. \tag{3}$$

As their name suggests, the local bounding cuts are locally-valid, namely that these cuts are valid only for the current node and its descendants in the enumeration tree, as they use the specific properties of the node. The local bounding cuts aim at favoring early locating the smallest edge cost at the subtree to help the solver concentrate on

finding a tour or proving the tour's non-existence in the subgraph restricted by $[l, u]$. Indeed, these cuts can tighten the bounds of the smallest edge cost l in the subproblems and thus narrow the interval $[l, u]$.

3.5 Variable fixing

To decrease the number of decision variables during the branch-and-cut algorithm, we fix some variables by adding corresponding cutting planes (e.g., add the inequality $x_e = 0$ if we want to fix x_e to 0). Naturally, variables that can not help to improve the incumbent solution should be fixed to 0. In this section, we propose two heuristics to determine such variables: one based on biconnected intervals compatible with costs and one based on fixed costs at tree nodes. Throughout this section, let $(\boldsymbol{x}, \overline{u}, \overline{l})$ be the current incumbent solution of the enumeration tree.

3.5.1 Biconnected-interval-based variable fixing

Using the same arguments as in Sect. 3.3, edges with costs C_i such that $\gamma(C_i) \geq \overline{u} - \overline{l}$ can not be part of tours that are better than the current best tour in terms of the max-min distance. Thus, the variables corresponding to such edges can be permanently fixed to 0 in the remaining nodes of the tree. Therefore, when a new incumbent solution $(\boldsymbol{x}, \overline{u}, \overline{l})$ is obtained, we add the following inequalities to LP relaxations

$$x_e = 0 \quad \forall e \in E : \gamma(c_e) \geq \overline{u} - \overline{l}. \tag{4}$$

Obviously, the inequalities (4) are valid for the remainder of the enumeration tree.

3.5.2 Fixed-costs-based variable fixing

The second heuristic to fix variables is due to the fact that each node of the enumeration tree is associated with two disjoint edge sets F_0 and F_1 where F_0, F_1 consist of edges that have been fixed to 0 and 1, respectively. Given a node $\langle F_0, F_1 \rangle$, let $\mathcal{H}'$ be a tour found by this node or its descendants. If the max-min distance of $\mathcal{H}'$ is smaller than the current best objective value (i.e., $\overline{u} - \overline{l}$), then the edges of $\mathcal{H}'$ belong to either F_1 or $\{e \in E : S_{C(F1)} - (\overline{u} - \overline{l}) < c_e < I_{C(F_1)} + (\overline{u} - \overline{l})\}$ where $S_{C(F_1)}$ and $I_{C(F_1)}$ are respectively the maximum and minimum of $C(F_1)$. Thus, edges that are not in either of these two sets can be fixed to 0 in this node and its descendant. The inequalities corresponding to the fixing of these variables are

$$x_e = 0, \quad \forall e \in E : c_e \notin (S_{C(F_1)} - (\overline{u} - \overline{l}), I_{C(F_1)} + (\overline{u} - \overline{l})) \tag{5}$$

Since the validity of inequalities (5) depends on fixed costs at the node, these inequalities are only valid for the considered node and its descendants.

3.6 Separation algorithms and strategies

In branch-and-cut algorithms, cutting planes are generated at some nodes of the tree by solving separation problems. An efficient branch-and-cut algorithm relies on not only good separation algorithms but also deft separation strategies. In this section, we present separation procedures and strategies for generating subtour elimination constraints and local bounding cuts. We first denote by $(\boldsymbol{x}^i, u^i, l^i)$ a fractional solution of an LP relaxation associated with a tree node.

3.6.1 Subtour elimination constraints

Recall that subtour elimination inequalities have the form $\sum_{e\in\delta(S)} x_e \geq 2$ where $S \subset V$. To find subtour elimination constraints (abbreviated as SECs in Fig. 1) violated by $\boldsymbol{x}^i$, one can construct a graph $G^i = (V, E^i)$ with edge set $E^i = \{e \in E \mid x_e^i > 0\}$. A weight associated with $e \in E^i$ is x_e^i. By this setting, a violated subtour elimination constraint is a cut whose weight is less than 2 in G^i. Such a cut can be found via a Gomory-Hu tree (Gomory and Hu 1961) of G^i, built from $|V|-1$ max-flow computation.

Since solving subtour's separation problem is computationally expensive and can provide no cutting planes, we only generate these inequalities at every 100 nodes instead of every node in the enumeration tree.

3.6.2 Local bounding cuts

At a node associated with ordered edge set pair $< F_0, F_1 >$, one can generate at most $O(|E|)$ local bounding cuts. However, if we generate all possible local bounding cuts at every node, the subproblems will be enormous and hard to solve. Thus, we only generate local bounding cuts with variables x_e such that $0 < x_e^i < 1$ and $I_{C(F_1)} < M_e$. In addition, since the local bounding cuts are mainly for the optimality phase, we only generate them when the IP relative gap is less than 0.5 at every 10 nodes.

4 Computational experiments

In this section, we present experimental results to assess the efficiency of our branch-and-cut algorithm. All the experiments are conducted on a PC Intel Core i7-10700 CPU 2.9GHz and 64 GB RAM. The algorithm is implemented in Python using CPLEX 12.10 with default setting and one solver thread. The CPU time limit for exploring the enumeration tree is set to 10,800 s. To make comparisons with the DT algorithms (Larusic and Punnen 2011), we use the identical testbed of 65 TSPLIB instances (Reinelt 1991). These instances include graphs with 14 to 493 vertices.

To verify the graph's biconnectivity in the lower bounding algorithm (i.e., Algorithm 1), we use a depth-first-search algorithm implemented by the Networkx algorithm (Hagberg et al. 2008).

To obtain initial tours for k-balanced algorithms, we use permutations of $\{1, 2, \ldots, |V|\}$ since the testing instances are complete graphs. The problem of com-

Table 3 The values of $\mathcal{K}$ correspond to instance sizes

Graph size ($\|V\|$)	$\|V\| < 50$	$50 \leq \|V\| < 100$	$100 \leq \|V\| < 200$	$\|V\| \geq 200$
$\mathcal{K}$	–	30	50	100

pleting a Hamiltonian cycle to find k-exchanges is resolved by MILP through using CPLEX 12.10. To initialize good upper bounds for the branch-and-cut algorithm, we execute k-balanced extreme and 3-balanced with 10 random tours. For instances with fewer than 50 vertices, only 3-balanced is performed. The k values are in the set $\{10, 20, \ldots, \mathcal{K}\}$ where $\mathcal{K}$ is defined based on instance sizes in Table 3. During the branch-and-cut algorithm, we execute k-balanced min/max with $k = \mathcal{K}$ and 3-balanced to improve the incumbent solutions.

We first select 12 instances from the test set to analyze the impact of our additional components in the branch-and-cut algorithm. The initial set comprises four small-sized instances (gr21, hk48, eil75, gr96), four medium-sized instances (pr136, si175, d198, tsp225) and four large-sized instances(a280, lin318, pcb442, d493). The first experiment in Sect. 4.1 aims at comparing our branch-and-cut algorithm to the commercial solver CPLEX 12.10, a general-purpose branch-and-cut algorithm. Then, Sect. 4.2 evaluates the impact of the components: local bounding cuts, the lower bounding algorithm, and the k-balanced algorithm. Finally, in Sect. 4.3, the entire testbed's results are shown with a comparison to the results of the DT algorithms reported in Larusic and Punnen (2011).

4.1 The effectiveness of the proposed branch-and-cut algorithm

In the first experiment, we compare our algorithm with the commercial solver CPLEX in solving (MILP-BTSP) specified in Sect. 2.

Table 4 reports the results of the two algorithms on the initial test set. Column *"Instance"* displays the instance's names where the number at the end stands for the number of graph vertices. The results of each algorithm contain the best objective value found by the algorithm within the CPU time limit (subcolumn *"Obj"*), the current IP relative gap (subcolumn *"Gap(%)"*), the running time in seconds (subcolumn *"CPU(s)"*), and the number of nodes in the enumeration tree (subcolumn *"Nodes"*). Notice that the running time includes the time spent on the initialization step and the enumeration tree exploration. Instances whose running times are marked with an asterisk (*) are instances that cannot be solved to proven optimality within the CPU time limit.

Numerical results illustrate that our branch-and-cut algorithm outperforms CPLEX (better performances are indicated by bold values). Indeed, within the CPU time limit, our algorithm can solve all 12 instances to proven optimality, whereas CPLEX can only solve 7 out of 12 instances to proven optimality. More precisely, CPLEX fails to prove the solution optimality for si175 and d198, and could not find an optimal solution for lin318, pcb442, and d493. Among the 12 instances, there is only one instance (gr21) on which our algorithm performs slower; for the rest, our algorithm solves the problems

Table 4 Comparison between the two algorithms on the 12 TSPLIB instances

Instance	Our B&C				CPLEX			
	Obj	Gap (%)	CPU (s)	Nodes	Obj	Gap (%)	CPU (s)	Nodes
gr21	115	0.0	0.6	0	115	0.0	0.2	298
hk48	156	0.0	4.3	157	156	0.0	5.7	3389
eil76	2	0.0	6.2	390	2	0.0	5241.6	470,000
gr96	314	0.0	93.9	1130	314	0.0	143.1	12,957
pr136	126	0.0	62.5	1126	126	0.0	243.8	15,709
si175	7	**0.0**	150.6	3854	7	28.6	10800.0*	113,245
d198	1122	**0.0**	2424.5	16,892	1122	58.7	10801.4*	62,677
tsp225	6	0.0	135.0	682	6	0.0	3955	28,550
a280	3	0.0	196.8	481	3	0.0	5319.6	31,856
lin318	31	**0.0**	499.3	1591	641	100.0	10804.8*	43,700
pcb442	27	**0.0**	9013.8	1592	283	100.0	10805.6*	22,712
d493	1193	**0.0**	4114.4	7399	1628	100.0	10808.1*	8935
Average		**0.0**	**1391.8**	**2941.2**		32.3	5744.1	67835.7

4 times faster on average than CPLEX. Furthermore, our algorithm's average tree size is 23 times smaller than that of CPLEX.

4.2 Impact of local cuts, lower bounding, and *k*-balanced components

The aim of this section is to evaluate the effectiveness of three additional components: local bounding cuts, the lower bounding algorithm, and the k-balanced algorithm. Toward this end, we create four algorithm variants. The *Full* variant is the complete version that uses all three components. The *Full x* variant represents a version excluding the component x, such as *Full Local cuts* omits local bounding cuts.

The computational results in Table 5 indicate that all proposed components are crucial for the efficiency of the branch-and-cut algorithm. Excluding any of these components can dramatically increase the algorithm's running time and lead to failure in solving several instances to proven optimality. We can order the impact of the components as follows: k-balanced > Lower bounding > Local cuts. More precisely, when the k-balanced algorithm is omitted, the computing time increases the most, i.e., 3.2 times compared to the Full version. Then, the absence of the lower bounding algorithm results in a slowdown of 2.7 times, and the lack of local bounding cuts leads to a 1.6 times decrease in speed. Instances whose running times are marked with an asterisk (*) are instances that cannot be solved to proven optimality by the variant in question within the CPU time limit. Bold values indicate the best performances among the tested algorithm variants.

4.3 Comparison to the double-threshold-based algorithms

Finally, we present the results of the branch-and-cut algorithm on the entire testbed with a comparison to the DT algorithms (Larusic and Punnen 2011). To ensure fairness in the comparisons, we reproduce the lower bounding algorithm and DT algorithms using the code furnished by the original authors.[1] Note that we only implement the modified double-threshold (MDT) algorithm and iterative bottleneck (IB) algorithm since these algorithms are reported to be the bests among the four heuristic variants proposed in Larusic and Punnen (2011). The CPU time limit for the DT algorithms is also set to 10,800 s.

Table 6 represents the algorithm's results. Column *"DT algorithms"* corresponds to the DT algorithms' results. Subcolumn *"LB"* reports lower bounds on the optimal value of the BTSP obtained by the procedure proposed in Larusic and Punnen (2011). Subcolumn *"Obj"* represents the best objective values found by the MDT or IB algorithm. Subcolumn *"CPU(s)"* gives the total time in seconds spent by the lower bounding procedure and the DT algorithms. In column *"Our B&C"* which represents the results of the proposed branch-and-cut algorithm, subcolumns *"Initial LB"* and *"Initial UB"* respectively indicate lower and upper bounds obtained by our lower bounding and k-balanced algorithms in the initialization step.

As shown in Table 6, the DT algorithms can find an optimal solution for 52 over 65 instances, but only 27 solutions are certified their optimality. In contrast, our branch-and-cut algorithm can solve to proven optimality 63 out of 65 instances within the CPU time limit, and thus provide optimality certificates of 36 solutions for the first time. Furthermore, for 13 of the 65 problems - mainly large-sized instances, our algorithm obtains solutions better than the DT algorithms. Although the two instances fl417 and pr439 can not be solved optimally within the time limit, their best objective values so far are significantly smaller than the DT algorithms' ones.

5 Conclusion

In this paper, we proposed a branch-and-cut algorithm for solving exactly the BTSP. We strengthened the branch-and-cut algorithm by local bounding cuts, edge elimination, and variable fixing. We also developed algorithms to produce good initial lower and upper bounds for the branch-and-cut algorithm. Several experiments on TSPLIB instances with less than 500 vertices are conducted. For 63 out of 65 instances, we obtained optimal solutions and for 13 of the 65 instances - mainly large-sized ones, our algorithm provided solutions with smaller objective values compared with the previous work in the literature (Larusic and Punnen 2011). For solving exactly large-scale instances of thousands of vertices, more mechanisms of tightening lower and upper bounds would be needed. Interesting directions for future research would be the investigation of new classes of local cuts and the improvement of the k-balanced algorithm.

[1] See at https://github.com/johnlarusic/arrow.

Table 5 Computational results of the algorithm variants

Instance	Full		Full local cuts		Full lower bounding		Full k-balanced	
	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)
gr21	0.0	0.6	0.0	0.5	0.0	0.6	0.0	0.6
hk48	0.0	4.3	0.0	3.8	0.0	3.8	0.0	16.5
eil76	0.0	6.2	0.0	471.3	0.0	1427.2	0.0	251.0
gr96	0.0	93.9	0.0	57.1	0.0	110.2	0.0	151.1
pr136	0.0	62.5	0.0	94.0	0.0	78.6	0.0	161.8
si175	0.0	150.6	0.0	3169.5	71.4	10806.9*	0.0	3579.2
d198	0.0	2424.5	0.0	1537.8	44.9	10810.3*	27.6	10824.9*
tsp225	0.0	135.0	0.0	991.6	0.0	3096.0	0.0	4232.0
a280	0.0	196.8	25.0	10826.9*	100.0	10825.6*	0.0	10074.2
lin318	0.0	499.3	0.0	461.8	0.0	1014.8	96.1	10835.7*
pcb442	0.0	9013.8	23.5	10899.8*	100.0	10858.9*	98.0	10847.4*
d493	0.0	4114.4	0.0	9118.0	0.0	6568.6	98.2	10862.8*
Average	**0.0**	**1391.8**	4.0	3136.0	26.4	4633.5	26.7	5153.1

Table 6 Results of the DT and branch-and-cut algorithms on 65 TSPLIB instances

Instance	DT algorithms (Larusic and Punnen 2011)			Our B&C			
	LB	Obj	CPU (s)	Initial LB	Initial UB	Obj	CPU (s)
burma14	120	134	0.02	120	134	**134**	0.3
ulysses16	837	868	0.05	173	868	**868**	0.5
gr17	94	119	0.07	80	129	**119**	0.5
gr21	110	115	0.07	65	120	**115**	0.6
ulysses22	837	868	0.29	157	868	**868**	0.6
gr24	33	33	0.1	33	45	33	0.7
fri26	21	21	0.04	21	25	21	0.5
bayg29	23	29	0.3	23	34	**29**	0.8
bays29	36	38	0.37	36	49	**38**	1.9
dantzig42	13	13	0.66	13	21	13	1.7
swiss42	14	14	0.22	14	32	14	1.7
att48	156	192	5.1	133	223	**190**$^{\downarrow}$	3.9
gr48	46	46	2.26	46	96	46	2.9
hk48	138	156	5.06	133	189	**156**	4.3
eil51	3	3	0.35	3	6	3	1.5
berlin52	139	149	5.33	113	151	**149**	5.5
brazil58	912	1124	12.51	912	1124	**1097**$^{\downarrow}$	7.7
st70	5	5	1.5	5	6	5	1.9
eil76	2	2	0.63	2	5	2	6.2
pr76	498	522	7.03	498	1015	**522**	8.6
gr96	281	314	83.51	281	561	**314**	93.9
rat99	5	5	2.84	5	9	5	9.2
kroA100	137	137	46.39	137	463	137	83

Table 6 continued

Instance	DT algorithms (Larusic and Punnen 2011)			Our B&C			
	LB	Obj	CPU (s)	Initial LB	Initial UB	Obj	CPU (s)
kroB100	129	145	47.96	129	471	**145**	65.7
kroC100	120	133	51.08	120	509	**133**	72.7
kroD100	140	140	42.54	137	269	140	422
kroE100	137	139	48.61	137	452	**139**	60.9
rd100	43	43	18.68	43	53	43	10.3
eil101	2	2	1.73	2	3	2	3.5
lin105	95	100	89.26	95	183	**100**	26.9
pr107	877	900	29.58	53	3645	**877**$^{\downarrow}$	25.2
gr120	27	31	20.39	27	94	**31**	67.9
pr124	364	408	258.94	364	731	**406**$^{\downarrow}$	93.2
bier127	2915	3023	186.82	874	3459	**2925**$^{\downarrow}$	29.8
ch130	18	22	25.91	17	60	**22**	56.7
pr136	103	126	29.95	103	1149	**126**	62.5
gr137	403	424	352.98	354	825	**424**	256.3
pr144	259	259	126	259	449	259	43
ch150	17	17	23.61	17	33	17	196.9
kroA150	89	91	120.43	89	452	**91**	122.2
kroB150	103	109	150.36	100	454	**109**	83.7
pr152	59	59	155.6	59	378	59	63.3
u159	142	142	38.08	135	822	142	1933.8
si175	7	7	50.76	5	21	7	150.6
brg180	0	0	0.31	0	0	0	2.8
rat195	4	4	8.58	4	16	4	499.6

Table 6 continued

Instance	DT algorithms (Larusic and Punnen 2011)			Our B&C			
	LB	Obj	CPU (s)	Initial LB	Initial UB	Obj	CPU (s)
d198	1105	1125	240.34	830	1355	**1122**↓	2424.5
kroA200	71	76	247.38	71	599	**76**	1607.7
kroB200	81	82	292.5	81	522	**82**	1242.7
gr202	778	875	1039.92	69	933	**787**↓	289.2
ts225	21	21	19.63	0	696	21	503.1
tsp225	6	6	50.05	6	21	6	135
pr226	450	504	1260.9	450	704	**504**	123.5
gr229	675	742	1737.24	622	1660	**706**↓	849.3
gil262	3	3	46.51	3	7	3	99.5
pr264	238	415	1226.01	238	3255	**340**↓	7386.6
a280	3	3	24.56	3	16	3	196.8
pr299	89	89	473.29	89	363	89	4258.6
lin318	31	31	430.65	31	133	31	499.3
rd400	11	11	269.08	11	17	11	243.3
fl417	199	260	4047.06	82	359	229↓	10931.2*
gr431	1943	2195	13038.11*	502	2876	**1962**↓	6555.5
pr439	810	1548	11967.86*	256	2583	994↓	11254.4*
pcb442	26	27	571.16	26	161	**27**	9013.8
d493	1191	1423	3136.15	34	1592	**1193**↓	4114.4

Instances with the bold objective value are solved to proven optimality for the first time, and instances with objective values marked by ↓ are ones that our algorithm can provide better solutions. For the instances fl417 and pr439, which can not be solved to proven optimality within the CPU time limit, their current IP relative gap are 64.2% and 74.3%, respectively

Funding The authors have not disclosed any funding.

Data availability Enquiries about data availability should be directed to the authors.

Declartions

Competing interests The authors have not disclosed any competing interests.

References

Arkin EM, Chiang Y-J, Mitchell JSB, Skiena SS, Yang T-C (1999) On the maximum scatter traveling salesperson problem. SIAM J Comput 29(2):515–544

Camerini PM, Maffioli F, Martello S, Toth P (1986) Most and least uniform spanning trees. Discrete Appl Math 15(2–3):181–197

Cappanera P, Scutella MG (2005) Balanced paths in acyclic networks: tractable cases and related approaches. Netw Int J 45(2):104–111

Galil Z, Schieber B (1988) On finding most uniform spanning trees. Discrete Appl Math 20:173–175

Gomory RE, Hu TC (1961) Multi-terminal network flows. J Soc Ind Appl Math 9(4):551–570

Hagberg AA, Schult DA, Swart PJ (2008) Exploring network structure, dynamics, and function using networks. In: Varoquaux G, Vaught T, Millman J (eds) Proceedings of the 7th python in science conference, Pasadena, CA USA, pp 11 – 15

Helsgaun K (2009) General k-opt submoves for the Lin–Kernighan TSP heuristic. Math Program Comput 1(2):119–163

Katoh N, Iwano K (1994) Efficient algorithms for minimum range cut problems. Networks 24:395–407

Larusic J, Punnen AP (2011) The balanced traveling salesman problem. Comput Oper Res 38(5):868–875

LaRusic J, Punnen AP (2014) The asymmetric bottleneck traveling salesman problem: algorithms, complexity and empirical analysis. Comput Oper Res 43:20–35

Lin S (1965) Computer solutions of the traveling salesman problem. Bell Syst Tech J 44(10):2245–2269

Martello S, Pulleyblank WR, Toth P, De Werra D (1984) Balanced optimization problems. Oper Res Lett 3(5):275–278

Plante RD, Lowe TJ, Chandrasekaran R (1987) The product matrix traveling salesman problem: an application and solution heuristic. Oper Res 35(5):772–783

Reinelt G (1991) TSPLIB—a traveling salesman problem library. ORSA J Comput 3(4):376–384

Scutellà MG (1998) A strongly polynomial algorithm for the uniform balanced network flow problem. Discrete Appl Math 81(1–3):123–131

Tarjan R (1972) Depth-first search and linear graph algorithms. SIAM J Comput 1(2):146–160

Turner L (2011) Variants of shortest path problems. Algorithmic Oper Res 6:91–104

Vairaktarakis GL (2003) On Gilmore–Gomory's open question for the bottleneck tsp. Oper Res Lett 31(6):483–491