A Branch-and-Cut algorithm for the Balanced Traveling Salesman Problem

Thi Quynh Trang VO, Mourad Baiou, Viet Hung Nguyen

Noname manuscript No.
(will be inserted by the editor)

A Branch-and-Cut algorithm for the Balanced Traveling Salesman Problem

Thi Quynh Trang Vo · Mourad Baiou · Viet Hung Nguyen*

Received: date / Accepted: date

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 in 2011 [9]. They proposed several heuristics based on the double-threshold framework, which converge to good-quality solutions though not always optimal (e.g. 27 provably optimal solutions were found among 65 TSPLIB instances of at most 500 vertices). In this paper, we design a special-purpose branch-and-cut algorithm for solving exactly 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 more and more precisely those edge costs. Other important ingredients of our algorithm are heuristics for improving the lower and upper bounds of the branch-and-bound tree. Experiments on the same TSPLIB instances show that our algorithm was able to solve to optimality 63 out of 65 instances.

Keywords Traveling Salesman Problem · Balanced optimization · Mixed-Integer Programming · Branch-and-Cut

Mathematics Subject Classification (2020) 90-10 · 90-05

T.Q.T. Vo, M. Baiou, V.H. Nguyen
INP Clermont Auvergne, Univ Clermont Auvergne, Mines Saint-Etienne, CNRS, UMR 6158 LIMOS, 1 Rue de la Chebarde, Aubiere Cedex, France
E-mail: thi_quynh_trang.vo@uca.fr, mourad.baiou@uca.fr, viet_hung.nguyen@uca.fr
* Corresponding author

1 Introduction

Given a finite set E with cost vector $\boldsymbol{c}$ and a family $\mathcal{F}$ of feasible subsets of E, the balanced optimization problem seeks a feasible subset $S^* \in \mathcal{F}$ that minimizes the difference in cost between the most expensive and least expensive element 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 et al. [12] 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 [16, 3], the balanced minimum cut [8], and the balanced spanning tree [4, 2].

In this paper, we consider the balanced version of the traveling salesman problem (TSP). In the context of the TSP, 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 balanced traveling salesman problem (BTSP) finds a tour in which the difference between the largest and smallest edge costs is minimum. 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 stated 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 as the problem of finding a Hamiltonian cycle in the graph can be reduced to the BTSP.

The BTSP was first studied by Larusic and Punnen [9] with applications in many practical problems, such as the nozzle guide vane assembly problem [13] and the cyclic workforce scheduling problem [17]. While most of the previous works about balanced optimization focused on polynomial-time algorithms, the BTSP was the first NP-hard case studied. 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 [12], widely used for balanced optimization problems. As its name suggests, the double-threshold algorithm maintains two thresholds of the edge costs of the tour: 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 $O(|V|^2)$ 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 [9] 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 [10] and the maximum scatter TSP [1]. With

these modifications, their algorithms produced provably optimal solutions for 27 out of 65 TSPLIB instances [14] from 14 to 493 vertices with a time limit of 18000 seconds per instance.

To the best of our knowledge, no exact algorithm based on Mixed-Integer Programming (MIP) for the BTSP has been proposed in the literature, although it is quite easy to formulate the BTSP through the existing MIP 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 heuristics for strengthening the lower and upper bounds of the BTSP. The efficiency of the proposed branch-and-cut algorithm is assessed through computational comparison to the double-threshold-based algorithms in [9]. Numerical results show that our algorithm can solve to optimality 63 instances out of 65 within 3 hours of CPU time.

The paper is organized as follows. In Section 2, we present a MIP formulation for the BTSP. Section 3 proposes a family of local cutting planes for the BTSP, called local bounding cuts. Then, the heuristics to improve the lower and upper bounds of the branch-and-bound tree are presented respectively in Sections 4 and 5. Section 6 describes the branch-and-cut algorithm used for the BTSP, which includes heuristics to eliminate edges and fix variables. Section 7 provides computational results to evaluate the algorithm's efficiency. Finally, we give some conclusions in Section 8.

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 endpoint in S, i.e., $\delta(S) = \{(i, j) \in E \mid i \in S \text{ and } j \in V \setminus S\}$. 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 MIP 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 MIP formulation for the BTSP as follows:

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

where $M_e = \min\{\max_{e' \in \delta(i)} c_{e'}, \max_{e' \in \delta(j)} c_{e'}\}$ for all $e \in E$. 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 ($x_e = 1$), inequalities (2e) read as $l \leq c_e$, which are true by the definition of l. Otherwise ($x_e = 0$), constraints (2e) become $l \leq M_e$, which are valid as $l \leq \max_{e \in \delta(i)} c_e, \forall i \in V$.

3 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 $(MIP - BTSP)$, 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 the linear programming (LP) relaxations and make solving the BTSP noticeably more time-consuming than solving the TSP. This can be seen in the following experiment. We addressed the TSP and BTSP on the TSPLIB instance si175 (with 175 vertices) by a general-purpose branch-and-cut algorithm with the same TSP constraints in the MIP formulations. While the TSP can be solved in 25 seconds, the BTSP can not be solved to optimality within 10800 seconds. Thus, the crucial point in solving the BTSP via branch-and-cut algorithms is

not the reinforcement of TSP constraints but the estimation of the largest and smallest edge costs. This section provides a family of local cutting planes to strengthen the bounds of the smallest edge cost in the LP relaxations.

Observe that in the branch-and-bound tree, each node is associated with an ordered pair $\langle F_0, F_1\rangle$ where $F_0, F_1 \subset E$ are two disjoint edge sets. Given a node $\langle F_0, F_1\rangle$, a tour found by the 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 $M^i_{C(F_1)}$ 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 $M^i_{C(F_1)}$. Based on this observation, we have the following inequalities, called *local bounding* cuts

$$l \leq c_e x_e + (1 - x_e) M^i_{C(F_1)} \quad \forall e \in E. \tag{3}$$

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 branch-and-bound 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 $[l, u]$.

4 Algorithm for improving the lower bound

A good lower bound enables to speed up branch-and-cut algorithms. Given a graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, we present below a heuristic partly inspired by the Hamiltonian verification procedure in [9] to find a lower bound of the BTSP.

As mentioned in [9], 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 heuristic 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 of the BTSP. Algorithm 1 gives a formal description of our lower bound heuristic. Before describing the heuristic in detail, we introduce some definitions and lemmas.

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.

Algorithm 1 Heuristic to find a lower bound of the BTSP

Input: A graph $G = (V, E)$ with edge costs $\boldsymbol{c}$.
Output: A lower bound of the BTSP.

Let $C_1 < C_2 < \cdots < C_p$ be the distinct costs of $\boldsymbol{c}$
$b_0 \leftarrow 1$, $C_{p+1} \leftarrow +\infty$
for $i \in [p]$ **do**
    $j \leftarrow b_{i-1}$
    while $j \leq p$ **do**
        if $G[C_j, C_{u_j}]$ is biconnected **then**
            $b_i \leftarrow j$
            break
        end if
        $j \leftarrow j + 1$
    end while
    if $j > p$ **then**
        $b_i \leftarrow p + 1$
    end if
end for
for $i \in [p]$ **do**
    $C_{l_i} \leftarrow C_1, C_{u_i} \leftarrow C_p$
    for $j \in [i]$ **do**
        if $C_{b_j} - C_j < C_{u_i} - C_{l_i}$ **then**
            $C_{l_i} \leftarrow C_j, C_{u_i} \leftarrow C_{b_j}$
        end if
    end for
end for
return $\min_{i \in [p]} C_{u_i} - C_{l_i}$.

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 $(MIP - BTSP)$, we have $\gamma^* \leq \mathcal{OPT}$.*

Thanks to Corollary 1, to obtain a lower bound 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 $b_j \in [p]$ (recall that $p = |C(E)|$) such that $G[C_j, C_{b_j}]$ is biconnected, for all $C_j \in C(E)$. 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. However, 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 [15]. In total, the complexity of Algorithm 1 is $O(|E|^2)$.

5 Local search algorithm to improve the upper bound

To improve the upper bound of the branch-and-cut algorithm, we develop a local search algorithm for the BTSP, called $k-$*balanced*, based on $k-$opt algorithms for the TSP [11, 7]. The algorithm takes a graph $G = (V, E)$ with edge costs $\boldsymbol{c}$ and an initial tour as input and returns an improved tour with a smaller max-min distance. We use $k-$balanced to provide a good feasible solution at the beginning of the branch-and-cut algorithm and enhance the incumbent solutions during the branch-and-cut.

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

Input: A tour $\mathcal{H}$ of G and a fixed number k.
Output: A tour with a smaller max-min distance.
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 $(\mathcal{H} \setminus F) \cup \overline{F}$ is a tour **then**
7: $\mathcal{H} \leftarrow (\mathcal{H} \setminus F) \cup \overline{F}$.
8: *improved* $\leftarrow$ **True**
9: **end if**
10: **end while**
11: **return** $\mathcal{H}$.

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 $\mathcal{H}' = (\mathcal{H} \setminus F) \cup \overline{F}$ 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 complete a tour from $\mathcal{H} \setminus F$, namely that $\overline{F} \subset EC(F, l', u')$. To construct $\overline{F}$, we solve the problem of completing a Hamiltonian cycle from $\mathcal{H} \setminus F$ 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 1 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*. Table 1 summarizes the three variants.

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 all min-cost edges. At step i, an edge f_i in $\mathcal{H} \setminus F$ 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

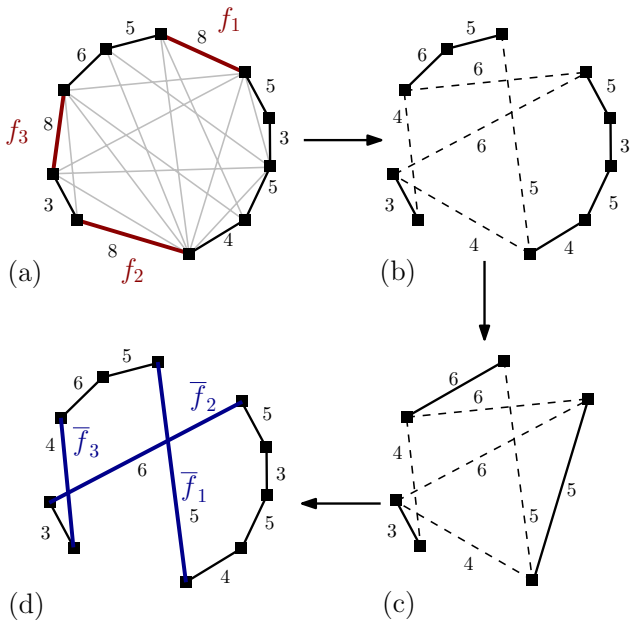

Fig. 1 Illustration of a 3−opt move in 3−balanced max. (1.a) represents a tour $\mathcal{H}$ whose largest and smallest edge costs are 8 and 3, respectively. We will remove all edges with max-cost 8 (f_1, f_2, f_3) from $\mathcal{H}$ and set $(l', u') = (l_{\mathcal{H}}, u_{\mathcal{H}} - 1) = (3, 7)$. (1.b) illustrates the remainder $\mathcal{H} \setminus F$ 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 $\mathcal{H} \setminus F$ 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.

	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 $d(c, \mathcal{H})$	$\min_{e \in \mathcal{H} \setminus F} c_e$	$\max_{e \in \mathcal{H} \setminus F} c_e$

Table 1 Selection rules of (F, l', u')

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. Let $d(c_e, \mathcal{H}) := \min(|l_{\mathcal{H}} - c_e|, |u_{\mathcal{H}} - c_e|)$ be a distance from a cost c_e to the edge costs of $\mathcal{H}$. We choose F as the set of k edges with the smallest distance $d(c, \mathcal{H})$. Then, (l', u') equals $(\min_{e \in \mathcal{H} \setminus F} c_e, \max_{e \in \mathcal{H} \setminus F} c_e)$. 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.

Algorithm 3 Selection rule for $k-$balanced min/max

Input: A graph $G = (V, E)$, a tour $\mathcal{H}$, a constant k, and an extreme type ET.
Output: (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: $f \leftarrow \arg\max_{e=(i,j) \in \mathcal{H}} \mid \delta(\{i, j\}) \cap \delta(F) \cap \{e \in E | l' \leq c_e \leq u'\} \mid$
8: $F \leftarrow F \cup \{f\}$
9: **end while**
10: **return** (F, l', u')

Algorithm 4 Selection rule for $k-$balanced extreme

Input: A graph $G = (V, E)$, a tour $\mathcal{H}$, and a constant k.
Output: (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: $removed\_cost \leftarrow \arg\min_{c_e \in C(\mathcal{H} \setminus F)} d(c, \mathcal{H})$
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')

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.

6 Branch-and-cut algorithm

In this section, we describe a branch-and-cut algorithm for solving exactly the BTSP. It contains mechanisms to locate the largest and smallest edge costs (i.e., local bounding cuts, edge elimination, and variable fixing) and algorithms to improve the lower and upper bounds.

The first step is to perform Algorithms 1 and 2 to yield a lower bound and an upper bound to start the branch-and-cut algorithm. These bounds are also used to eliminate edges and reduce the formulation's size. Details are given in Section 6.1.

After the initialization steps, the algorithm constructs a search tree (a.k.a branch-and-bound tree) whose root node is the LP relaxation of $(MIP-BTSP)$ without subtour elimination constraints. When an integer solution is found, violated subtour constraints are found and added to the formulation. If this solution satisfies all subtour constraints and has the best objective value, it is called the incumbent solution. When obtaining a new incumbent solution, Algorithm 2 is called to enhance this solution and decrease the upper bound of the branch-and-cut algorithm. At nodes in which the solutions to the subproblems are fractional, local bounding cuts and subtour constraints are generated following the separation strategies presented in Section 6.3. To accelerate exploring the nodes, we integrate into the branch-and-cut algorithm several variable fixing techniques, which are described in Section 6.2. Other fundamental components, such as node and variable selections, follow the default rules of the commercial solver CPLEX 12.10.

The algorithm is sketched as follows:

Step 0: *(Initialization)*
0.1 Run Algorithms 1 and 2 to get a lower bound of the BTSP and an initial feasible solution $(\boldsymbol{x}^0, l^0, u^0)$, respectively.
0.2 *(Edge elimination)* Eliminate edges based on $(\boldsymbol{x}^0, l^0, u^0)$ following Section 6.1.
0.3 Let $\mathcal{N}$ be the node set of the branch-and-bound tree and $(\overline{\boldsymbol{x}}, \overline{l}, \overline{u})$ be the current incumbent solution. Initialize $\mathcal{N}$ by the LP relaxation of $(MIP - BTSP)$ without (2c) and $(\overline{\boldsymbol{x}}, \overline{l}, \overline{u})$ by $(\boldsymbol{x}^0, l^0, u^0)$.
Step 1: *(Node selection)* If $\mathcal{N}$ is empty, then return $(\overline{\boldsymbol{x}}, \overline{l}, \overline{u})$ and terminate. Otherwise, take out a subproblem $\mathcal{P}$ from $\mathcal{N}$.
Step 2: Solve $\mathcal{P}$. If $\mathcal{P}$ is infeasible, go to Step 1. Otherwise, let $(\boldsymbol{x}^*, l^*, u^*)$ be an optimal solution to $\mathcal{P}$.
Step 3: If $u^* - l^* \geq \overline{u} - \overline{l}$, go to Step 1.
Step 4: If $\boldsymbol{x}^*$ is integer, verify subtour elimination constraints with $\boldsymbol{x}^*$.
4.1 If $\boldsymbol{x}^*$ satisfies subtour constraints, then $(\boldsymbol{x}^*, l^*, u^*)$ is a new incumbent solution. Run Algorithm 2 to improve $(\boldsymbol{x}^*, u^*, l^*)$. Replace $(\overline{\boldsymbol{x}}, \overline{l}, \overline{u})$ by $(\boldsymbol{x}^*, l^*, u^*)$, fix variables as in Section 6.2 and go to Step 1.
4.2 Otherwise, add subtour elimination constraints violated by $\boldsymbol{x}^*$ to the formulation and go to Step 2.
Step 5: *(Cut generation)* Generate violated valid inequalities by the separation strategies in Section 6.3 and fix variables by the heuristics introduced in Section 6.2.
Step 6: *(Variable selection)* Choose a fractional variable to branch. Add the two resulting subproblems to $\mathcal{N}$ and go to Step 1.

6.1 Edge elimination

To reduce the formulation's size and accelerate solving the LP relaxations, we eliminate edges that can not occur in the optimal tour of the BTSP. Remember that the branch-and-cut algorithm aims to improve the incumbent solution more and more. Thus, if we can prove that the occurrence of an edge leads to a tour worse than the incumbent tour, we can remove this edge from the formulation.

Let $\mathcal{H}^0$ be the initial tour found by Algorithm 2. 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. Then, edges with costs C_i satisfying $\gamma(C_i) > u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$ can not be a part of the 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 $\gamma(c_e) > u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$.

6.2 Variable fixing

Besides eliminating edges at the beginning, we also fix variables during the branch-and-cut algorithm to decrease the number of variables to be controlled and tighten the LP relaxations. Naturally, variables that cannot help to improve the incumbent solution should be fixed to 0. To fix variables, we add the inequalities corresponding to the fixing of the variables as cutting planes. This section proposes two heuristics to determine variables that can be fixed to 0: one based on the biconnected intervals and one based on fixed costs at nodes. Throughout this section, we denote by $(\overline{\boldsymbol{x}}, \bar{l}, \overline{u})$ the current incumbent solution of the search tree.

6.2.1 Biconnected-interval-based variable fixing

Using the same arguments as in Section 6.1, edges with costs C_i such that $\gamma(C_i) \geq \overline{u} - \bar{l}$ can not appear in solutions that are better than the incumbent solution. Thus, such edges can be permanently fixed to 0 in the remaining nodes of the branch-and-bound tree. In particular, when a new incumbent solution $(\overline{\boldsymbol{x}}, \bar{l}, \overline{u})$ is found, we add the following inequalities to the formulation

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

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

6.2.2 Fixed-costs-based variable fixing

The second heuristic to fix variables is due to the fact that each node of the search 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 $\langle F_0, F_1 \rangle$, we respectively denote by $M^i_{C(F_1)}$ and $M^s_{C(F_1)}$ the minimum and maximum of $C(F_1)$. Let $\mathcal{H}'$ be a tour that has the max-min distance

smaller than the incumbent solution's one and is found by the node or its descendants. Obviously, $\mathcal{H}'$ only comprises the edges of F_1 and edges with costs in $(M^s_{C(F_1)} - (\overline{u} - \overline{l}), M^i_{C(F_1)} + (\overline{u} - \overline{l}))$. The remaining edges, which do not satisfy the above cost condition, can be fixed to 0. The inequalities corresponding to the fixing of these variables are

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

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.

6.3 Separation algorithms and strategies

An efficient branch-and-cut algorithm relies on good separation algorithms and deft separation strategies. We propose here separation procedures and strategies for subtour constraints and local bounding cuts. We first denote by $(\boldsymbol{x}^*, u^*, l^*)$ a fractional solution at a node of the branch-and-bound tree.

6.3.1 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 constraints violated by $\boldsymbol{x}^*$, one can construct a graph $G^* = (V, E^*)$ with edge set $E^* = \{e \in E \mid x^*_e > 0\}$. A cost associated with $e \in E^*$ is x^*_e. By this setting, a violated subtour constraint is a cut whose weight is less than 2 in G^*. Such a cut can be found via a Gomory-Hu tree [5] of G^*, built from $|V| - 1$ max-flow computation.

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

6.3.2 Local bounding cuts

At a node of the branch-and-bound tree, one can generate at most $O(|E|)$ local bounding cuts. If we generate all possible local bounding cuts at every node, the subproblems will be enormous and very hard to solve. Thus, we only generate local bounding cuts with variable x_e such that $x^*_e > 0$ and $M_e < M^i_{C(F_1)}$. In addition, since the local bounding cuts are mainly for the optimality phase, we only generate them when the MIP relative gap is less than 0.5 at every 10 nodes.

7 Computational experiments

In this section, we conduct some experiments 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 with 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 branch-and-bound tree is 10800 seconds (3 hours) per instance. For the testbed, we use the same TSPLIB instances [14] from 14 to 493 vertices as [9].

The biconnectivity verification problem in Algorithm 1 has been solved using the Networkx package [6]. For the $k-$balanced algorithms, since the test instances are completed graphs, we use permutations of $\{1, \ldots, |V|\}$ to initialize tours. The problem of completing a Hamiltonian cycle to find $k-$exchanges is solved by integer programming. To find a good upper bound, we run $k-$balanced with 10 random tours; at each iteration, we launch $k-$balanced extreme with k in $\{0, 10, \ldots, \mathcal{K}\}$ and $3-$balanced (if possible). To enhance the incumbent solutions during the branch-and-cut algorithm, we run $k-$balanced min and $k-$balanced max with $k = \mathcal{K}$ and $3-$balanced. The value of $\mathcal{K}$ is defined in Table 2.

Graph size ($\|V\|$)	$\|V\| < 50$	$50 \leq \|V\| < 100$	$100 \leq \|V\| < 200$	$\|V\| \geq 200$
K	0	30	50	100

Table 2 The value of $\mathcal{K}$ corresponds to graph sizes.

We first selected 12 instances from the test set to analyze the impact of ingredients in our 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 Section 7.1 aims at comparing our branch-and-cut algorithm to the commercial solver CPLEX 12.10. Then, Section 7.2 analyzes the impact of the components: local bounding cuts, Algorithm 1 to find a lower bound and Algorithm 2 to improve the upper bound. Finally, in Section 7.3, the entire testbed's results are shown with a comparison to the results of the double-threshold-based algorithms in [9].

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

In the first experiment, we compare our algorithm with the commercial solver CPLEX for solving the formulation ($MIP - BTSP$) specified in Section 2.

Table 3 reports the results of the two algorithms on the initial test set. Column *"Size"* indicates the number of vertices of instances, which are equal to the numbers in instances' names. The results of each algorithm in the table contain the objective value (labeled *"Obj"*), the running time in seconds (labeled *"Time(s)"*), and the number of nodes in the search tree (labeled *"Nodes"*). Notice that the running time includes the time spent on the initialization steps and the search tree exploration. Instances whose running times are marked with an asterisk (*) are instances that cannot be solved to optimality within the CPU time limit, and their reported objective value is the best one found so far.

Instance	Size	Our B&C			CPLEX		
		Obj	Time(s)	Nodes	Obj	Time(s)	Nodes
gr21	21	115	0.6	0	115	0.2	298
hk48	48	156	4.3	157	156	5.7	3389
eil76	76	2	6.2	390	2	5,241.6	470000
gr96	96	314	93.9	1130	314	143.1	12957
pr136	136	126	62.5	1126	126	243.8	15709
si175	175	7	150.6	3854	7	10,801.0*	113245
d198	198	1122	2424.5	16892	1122	10,801.4*	62677
tsp225	225	6	135.0	682	6	3,955.0	28550
a280	280	3	196.8	481	3	5,319.6	31856
lin318	318	**31**	499.3	1591	641	10,804.8*	43700
pcb442	442	**27**	9013.8	1592	283	10,805.6*	22712
d493	493	**1193**	4114.4	7399	1628	10,808.1*	8935
Average			**1,391.8**	**2,941.2**		5,744.2	67,835.7

Table 3 Comparison between the two algorithms on 12 TSPLIB instances

Numerical results illustrate that our branch-and-cut algorithm outperforms CPLEX. Indeed, our algorithm can rapidly solve all 12 instances within the time limit, whereas CPLEX can solve only 7 out of 12 cases. In detail, CPLEX fails to prove the solution optimality for instances si175, d198 and find the optimal solutions for instances lin318, pcb442, 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 4 times faster on average than CPLEX. Moreover, our algorithm's average tree size is 23 times smaller than that of CPLEX.

7.2 Impact of the local cuts, lower bound and upper bound components

In this section, we aim to analyze the effectiveness of the three key components: local bounding cuts, the lower bound algorithm, and the upper bound algorithm. Four algorithm variants are created for this purpose. The first setting *Full* corresponds to the full version, which uses all components. The setting *Full* x represents the version excluding the component x, e.g., the setting *Full Local cuts* is the version without local bounding cuts.

The computational results in Table 4 show that all components play important roles in the branch-and-cut algorithm. Excluding one of the components from the algorithm substantially raises the running time and makes the algorithm cannot solve several instances to optimality within the time limit. When using all components, the running time decreases by a factor of 3. We can order the effectiveness of the components as follows: Upper bound > Lower bound > Local cuts. The upper bound component yields the most improvement on the CPU time (3.2 times faster), then the lower bound component (2.7 times) and valid inequalities (1.6 times).

For a deeper analysis, we present in Table 5 the lower and upper bounds obtained by our algorithm and CPLEX. It can be seen that the lower and upper bounds found by our algorithm are extremely sharper than CPLEX's ones. Furthermore, the time spent finding the upper bounds of our algorithm on average is also smaller than that of CPLEX.

Instance	Full		Full Local cuts		Full Lower bound		Full Upper bound	
	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)	Nodes
gr21	0.6	0	0.5	0	0.6	15	0.6	668
hk48	4.3	157	3.8	280	3.8	213	16.5	5544
eil76	6.2	390	471.3	116330	1427.2	179590	251.0	45520
gr96	93.9	1130	57.1	6835	110.2	9586	151.1	11161
pr136	62.5	1126	94.0	4691	78.6	3085	161.8	9320
si175	150.6	3854	3169.5	103012	10806.9*	144123	3579.2	77106
d198	2,424.5	16892	1537.8	61366	10810.3*	99761	10824.9*	41125
tsp225	135.0	682	991.6	22267	3096.0	37827	4232.0	31270
a280	196.8	481	10826.9*	52096	10825.6*	147408	10074.2	47577
lin318	499.3	1591	461.8	1476	1014.8	6832	10835.7*	45920
pcb442	9,013.8	1592	10899.8*	16840	10858.9*	3401	10847.4*	7621
d493	4,114.4	7399	9118.0	14640	6568.6	4381	10862.8*	3100
Average	**1,391.8**	**2,941.2**	3,136.0	33,319.4	4,633.5	53,018.5	5,153.1	27,161.0

Table 4 Computational results of the algorithm variants

Instance	Our B&C				CPLEX			
	LB	LB Time	UB	UB Time	LB	LB Time	UB	UB Time
gr21	65	0.03	120	0.48	0	0.00	714	0.06
hk48	133	0.53	189	2.42	0	0.01	2612	0.06
eil76	2	0.06	5	1.23	0	0.01	60	0.04
gr96	281	6.30	561	5.04	0	0.02	5864	0.41
pr136	103	2.00	1149	3.23	0	0.06	12657	0.2
si175	5	0.82	21	5.70	0	0.06	303	1.92
d198	830	24.17	1355	9.23	0	0.08	2757	2.67
tsp225	6	1.57	21	14.76	0	0.48	494	16.78
a280	3	1.65	16	23.08	0	1.23	171	50.59
lin318	31	34.42	133	41.28	0	0.91	2929	45.89
pcb442	26	45.71	161	54.82	0	16.82	3790	3106.86
d493	34	57.87	1592	246.71	0	13.17	2947	1051.14
Average	**132.18**	15.92	**473.00**	**37.05**	0	**2.99**	3144.00	388.78

Table 5 Lower and upper bounds provided by the two algorithms

7.3 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 double-threshold-based (DT-based) algorithms introduced in [9], i.e., the modified double-threshold (MDT) and iterative bottleneck (IB) algorithms.

In Table 6, column "DT-based algorithm" reports the results of the DT-based algorithms provided by [9]. Subcolumn *"Obj"* represents the best objective value found by the MDT or IB algorithm. Subcolumn *"Opt?"* indicates whether the solution is provably optimal, namely that the lower bound equals the objective value. Subcolumn *"Time"* gives the total time for calculating the lower bound and solving the instance by the MDT or IB algorithm. Notice that the running time of the DT-based algorithms as reported in [9] and recopied in Table 6 is measured with experiment settings differing from ours, i.e the algorithms are coded in C programming language and tested on a PC with 3.40 GHz Pentium 4 CPU and 2 GB of RAM, and the time limit is 18000 seconds. We present the running time here not for comparison purposes but for reference only. As reported in [9], the DT-based algorithms converged to solutions within 10% optimality estimated based on lower bound values, in which 27 solutions are provably optimal. The found solutions are the best solutions that can be

Instance	DT-based algorithm [9]				Our B&C				
	LB	Obj	Time	Opt?	LB	UB	Obj	Time	Nodes
burma14	120	134	0.2		120	134	**134**	0.3	0
ulysses16	837	868	0.4		173	868	**868**	0.5	0
gr17	94	119	0.1		80	129	**119**	0.5	8
gr21	110	115	0.1		65	120	**115**	0.6	0
ulysses22	837	868	1.7		157	868	**868**	0.6	0
gr24	33	33	0.1	yes	33	45	33	0.7	0
fri26	21	21	0.1	yes	21	25	21	0.5	0
bayg29	23	29	0.3		23	34	**29**	0.8	17
bays29	36	38	0.3		36	49	**38**	1.9	642
dantzig42	13	13	0.2	yes	13	21	13	1.7	140
swiss42	14	14	0.4	yes	14	32	14	1.7	179
att48	156	192	14.1		133	223	**190**$^{\downarrow}$	3.9	303
gr48	46	46	2.3	yes	46	96	46	2.9	173
hk48	138	156	9.8		133	189	**156**	4.3	157
eil51	3	3	0.3	yes	3	6	3	1.5	10
berlin52	139	151	11.5		113	151	**149**$^{\downarrow}$	5.5	573
brazil58	912	1125	19.8		912	1124	**1097**$^{\downarrow}$	7.7	264
st70	5	5	1.9	yes	5	6	5	1.9	59
eil76	2	2	1.1	yes	2	5	2	6.2	390
pr76	498	522	25.5		498	1015	**522**	8.6	186
gr96	281	314	941.1		281	561	**314**	93.9	1130
rat99	5	5	3.1	yes	5	9	5	9.2	333
kroA100	137	137	93.2	yes	137	463	137	83	1200
kroB100	129	145	111.1		129	471	**145**	65.7	917
kroC100	120	133	136.0		120	509	**133**	72.7	2500
kroD100	140	140	67.2	yes	137	269	140	422	4811
kroE100	137	139	173.5		137	452	**139**	60.9	865
rd100	43	43	23.9	yes	43	53	43	10.3	205
eil101	2	2	2.4	yes	2	3	2	3.5	12
lin105	95	100	217.4		95	183	**100**	26.9	221
pr107	877	877	84.4	yes	53	3645	877	25.2	1007
gr120	27	31	50.4		27	94	**31**	67.9	1174
pr124	364	411	500.4		364	731	**406**$^{\downarrow}$	93.2	949
bier127	2915	3084	493.8		874	3459	**2925**$^{\downarrow}$	29.8	106
ch130	18	22	36.5		17	60	**22**	56.7	827
pr136	103	126	58.8		103	1149	**126**	62.5	1126
gr137	403	428	3,239.7		354	825	**424**$^{\downarrow}$	256.3	2647
pr144	259	259	347.8	yes	259	449	259	43	333
ch150	17	17	18.6	yes	17	33	17	196.9	520
kroA150	89	91	330.4		89	452	**91**	122.2	1279
kroB150	103	109	356.0		100	454	**109**	83.7	708
pr152	59	59	230.5	yes	59	378	59	63.3	1326
u159	142	142	111.0	yes	135	822	142	1933.8	42815
si175	7	7	0.0	yes	5	21	7	150.6	3854
brg180	0	0	0.7	yes	0	0	0	2.8	0
rat195	4	4	16.7	yes	4	16	4	499.6	3920
d198	1105	1140	391.8		830	1355	**1122**$^{\downarrow}$	2424.5	16892
kroA200	71	76	660.3		71	599	**76**	1607.7	2050
kroB200	81	82	620.9		81	522	**82**	1242.7	4070
gr202	778	927	4,813.1		69	933	**787**$^{\downarrow}$	289.2	241
ts225	0	21	50.9		0	696	**21**	503.1	6148
tsp225	6	6	88.7	yes	6	21	6	135	682
pr226	450	504	1,575.3		450	704	**504**	123.5	0
gr229	675	742	14,936.8		622	1660	**706**$^{\downarrow}$	849.3	230
gil262	3	3	69.5	yes	3	7	3	99.5	110
pr264	238	415	3,132.9		238	3255	**340**$^{\downarrow}$	7386.6	5589
a280	3	3	49.5	yes	3	16	3	196.8	481
pr299	89	89	1,173.6	yes	89	363	89	4258.6	476
lin318	31	31	1,442.0	yes	31	133	31	499.3	1591
rd400	11	11	491.7	yes	11	17	11	243.3	171
fl417	199	317	2,318.3		82	359	229$^{\downarrow}$	10931.2*	93800
gr431	1943	2230	42,966.3*		502	2876	**1962**$^{\downarrow}$	6555.5	11805
pr439	810	1620	5,973.9		256	2583	994$^{\downarrow}$	11254.4*	36687
pcb442	26	27	1,302.6		26	161	**27**	9013.8	1592
d493	1191	1459	9,416.0		34	1592	**1193**$^{\downarrow}$	4114.4	7399

Table 6 Numerical results of the Branch-and-Cut algorithm on 65 TSPLIB instances. Instances with the bold objective value are solved to optimality for the first time and instances with objective values marked by $\downarrow$ are ones that our algorithm can produce better solutions.

found by the algorithms without regard to the CPU time limit, except for the instance gr431.

Table 6 shows that our algorithm can solve to optimality 63 out of 65 instances within the time limit (10800 seconds), in which 36 instances are solved to optimality for the first time. For 14 of the 65 problems - mainly large-sized instances, our algorithm obtains solutions better than the DT-based 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-based algorithms' ones.

8 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 heuristics to improve the lower and upper bounds of the 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 14 of the 65 instances - mainly large-sized ones, our algorithm provided solutions with smaller objective values comparing with the previous work in the literature [9]. For solving exactly large scale instances of thousands vertices, more mechanisms of tightening lower and upper bounds would be needed. Interesting directions for future research would be the investigation for new classes of local cuts and the improvement of the $k-$balanced algorithm.

References

1. Esther M Arkin, Yi-Jen Chiang, Joseph SB Mitchell, Steven S Skiena, and Tae-Cheon Yang. On the maximum scatter traveling salesperson problem. *SIAM Journal on Computing*, 29(2):515–544, 1999.
2. Paolo M Camerini, Francesco Maffioli, Silvano Martello, and Paolo Toth. Most and least uniform spanning trees. *Discrete Applied Mathematics*, 15(2-3):181–197, 1986.
3. Paola Cappanera and Maria Grazia Scutella. Balanced paths in acyclic networks: Tractable cases and related approaches. *Networks: An International Journal*, 45(2):104–111, 2005.
4. Zvi Galil and Baruch Schieber. On finding most uniform spanning trees. *Discrete Applied Mathematics*, 20, 1988.
5. Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.
6. Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.
7. Keld Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2):119–163, 2009.
8. Naoki Katoh and Kazuo Iwano. Efficient algorithms for minimum range cut problems. *Networks*, 24, 1994.
9. John Larusic and Abraham P Punnen. The balanced traveling salesmanproblem. *Computers & Operations Research*, 38(5):868–875, 2011.

10. John LaRusic and Abraham P Punnen. The asymmetric bottleneck traveling salesman problem: algorithms, complexity and empirical analysis. *Computers & Operations Research*, 43:20–35, 2014.
11. Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.
12. Silvano Martello, William R Pulleyblank, Paolo Toth, and Dominique De Werra. Balanced optimization problems. *Operations Research Letters*, 3(5):275–278, 1984.
13. Robert D Plante, Timothy J Lowe, and R Chandrasekaran. The product matrix traveling salesman problem: an application and solution heuristic. *Operations Research*, 35(5):772–783, 1987.
14. Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.
15. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
16. Lara Turner. Variants of shortest path problems. *Algorithmic Oper. Res.*, 6, 2011.
17. George L Vairaktarakis. On gilmore–gomory's open question for the bottleneck tsp. *Operations Research Letters*, 31(6):483–491, 2003.