Knowledge-Guided Local Search for the Prize-collecting Steiner Tree Problem in Graphs

Zhang-Hua Fu [a,b] and Jin-Kao Hao [b,c,*]

[a] *Robotics Laboratory for Logistics Service, Institute of Robotics and Intelligent Manufacturing, The Chinese University of Hong Kong, Shenzhen, 518172, Shenzhen, China*

[b] *LERIA, Université d'Angers, 2 Bd Lavoisier, 49045 Angers Cedex 01, France*

[c] *Institut Universitaire de France, 1 rue Descartes, 75231 Paris Cedex 05, France*

Knowledge-Based Systems, 2017, DOI: 10.1016/j.knosys.2017.04.010

Abstract

The *prize-collecting Steiner tree problem in graphs (PCSPG)*, as well as its rooted variant (*RPCST*), are target problems of the 11th DIMACS (the Center for Discrete Mathematics and Theoretical Computer Science) Implementation Challenge held in collaboration with ICERM (the Institute for Computational and Experimental Research in Mathematics). To solve these two problems, this paper proposes a knowledge-guided local search algorithm (K-ILS) [1], which integrates dedicated search strategies and explores structure information of problem instances. K-ILS uses an effective swap-vertex operator for tree transformation associated with a discriminating auxiliary evaluation function as well as several knowledge-guided perturbation strategies. K-ILS additionally employs two new path-based move operators to generate neighboring solutions. The computational results achieved on the benchmark instances of the 11th DIMACS Implementation Challenge using the same computing platform and competition rules demonstrate that K-ILS performs very well compared to the leading algorithms of the challenge. We report additional experiments to analyze the impact of the key components to the performance of the proposed algorithm.

Keywords: Prize-collecting Steiner tree problem; network design and optimization; knowledge-guided local search; tree transformation operators.

[1] This paper extends [15], which was presented at the workshop of the 11th DIMACS Implementation Challenge, but has not been formally published.

* Corresponding author.
Email addresses: fu@info.univ-angers.fr (Zhang-Hua Fu),

1 Introduction

Given an undirected graph $G(V, E)$ with a set V ($|V| = n$) of vertices and a set E ($|E| = m$) of edges, each vertex $i \in V$ is associated with a real-valued prize $p_i \geq 0$ (vertex i is a customer vertex if $p_i > 0$, and a non-customer or Steiner vertex otherwise), and each edge $e \in E$ is associated with a real-valued cost $c_e \geq 0$, then the Prize-collecting Steiner Tree Problem in graphs (PCSPG) involves finding a subtree T (with vertex set V_T and edge set E_T respectively) of G, so as to minimize the sum of the costs of its edges plus the prizes of the vertices not spanned by T, i.e., [28]:

$$Minimize\ f(T) = \sum_{e \in E_T} c_e + \sum_{i \notin V_T} p_i. \tag{1}$$

In addition to this basic problem, the PCSPG has some other closely related variants, such as the rooted version of the PCSPG (named RPCST for short), which considers an additional source vertex as the root which must be part of any feasible solution. Moreover, the classical Steiner tree problem in graphs (SPG) [25] is a particular case of the PCSPG, if each terminal of the graph is associated with a high enough prize, and each Steiner vertex is associated with a prize equals to zero.

The PCSPG, as well as its related variants, are known to be relevant models to formulate many important network design problems. Meanwhile, given that the classical SPG is NP-hard [30], the PCSPG is at least as difficult and computationally challenging as the SPG in the general case. Recognizing their theoretical importance and wide practical applications, the 11th DIMACS Implementation Challenge in collaboration with ICERM (from June 2013 to December 2014) was dedicated to the class of broadly defined Steiner tree problems including both the PCSPG and the RPCST [29].

The PCSPG (along with the RPCST) has been extensively investigated since it was proposed [11,37] (where it firstly appeared as the so-called *node weighted Steiner tree problem*). Many approaches have been proposed for this problem, which could be mainly classified into three categories: approximation algorithms which aim to find solutions with provable quality, exact algorithms which warrant optimality of the obtained solutions, and heuristics which seek high-quality solutions within a reasonable time frame. Moreover, effective reduction tests like those introduced in [38] were usually used as a pre-processing technique to transform the original graph to an equivalent reduced graph.

Among the approximation approaches, Bienstock et al. proposed a first 3-

jin-kao.hao@univ-angers.fr (Jin-Kao Hao).

approximation algorithm [6]. Later, Goemans and Williamson [20] used a primal-dual scheme to derive a $(2 - \frac{1}{n-1})$-approximation for the RPCST in $O(n^2 \log n)$. By trying all possible choices for the root, they obtained a $(2 - \frac{1}{n-1})$-approximation for the PCSPG with a complexity of $O(n^3 \log n)$ [21]. Furthermore, Johnson et al. showed a $(2 - \frac{1}{n-1})$-approximation algorithm for the PCSPG with $O(n^2 \log n)$ running time [28], which was subsequently extended to the RPCST in [35]. The new algorithm by Feofiloff et al. [12] achieved an approximation ratio of $(2 - \frac{2}{n})$ within $O(n^2 \log n)$ time. Finally, Archer et al. [3] reported the best approximation ratio with an upper bound of 1.9672.

Several exact approaches (mainly based on different integer programming formulations) have been proposed for the PCSPG. Lucena and Resende [34] presented a polyhedral cutting plane based algorithm, which solves optimally 96 of the 114 classical test instances (with up to 1000 vertices and 25000 edges). Ljubić et al. [33] re-formulated the PCSPG in a directed formulation and implemented a branch-and-cut algorithm which yields outstanding results (with all the 189 test instances solved to optimality, including 35 real-world instances with up to 1825 vertices and 214095 edges). Salles Da Cunha et al. [36] used a Lagrangian non delayed relax and cut (NDRC) algorithm to generate primal and dual bounds to the problem. Experimental results on a new group of difficult instances showed its competitiveness for solving the PCSPG. Meanwhile, no exact algorithm in the literature is able to optimally solve all the existing PCSPG benchmark instances. This is especially the case for a number of particularly difficult instances newly introduced for the 11th DIMACS Challenge.

Given the NP-hard nature of the PCSPG, heuristics naturally constitute another important solution approach. For instance, Canuto et al. [7] described a multi-start local search algorithm for the PCSPG, which uses a primal-dual algorithm to generate initial solutions, two basic move operators (add or remove a vertex) to obtain neighboring solutions, a path-relinking procedure to create intermediate solutions between two elite solutions, followed by a post-optimization procedure using variable neighborhood search. Klau et al. [31] developed a complex algorithmic framework, which first applies an extensive pre-processing procedure to reduce the given graph without changing the optimal solution. The reduced problem is then solved by a memetic search algorithm which combines a steady-state evolutionary algorithm and an exact subroutine for the problem on trees. The solution is finally improved by an integer linear programming (ILP) based post-optimization subroutine. Goldbarg et al. [23] presented another hybrid algorithm, which uses a transgenetic algorithm (based on several coding, evolution and updating strategies) to search good solutions, and then applies a path-relinking procedure for further improvement. Biazzo et al. [5] introduced an approach derived from the cavity method, based on the zero temperature limit of the cavity equations, to form a simple (a fixed point equation solved iteratively) and parallelizable algorithm.

Very recently, Akhmedov et al. [1] presented a divide-and-conquer matheuristic method, which is composed of a pre-processing procedure, a heuristic clustering algorithm and an exact solver. This algorithm was tested on a number of huge PCSPG instances transformed from biological graphs with special structures, and showed good performances for these specific instances.

The 11th DIMACS Implementation Challenge has attracted a number of new algorithms, including approximation algorithms [26], exact algorithms [2, 10, 18], heuristics [5, 15] and hybrid algorithms [13]. These algorithms definitively advanced the state of the art on the RPCST and PCSPG. For each existing problem instance, the previous best-known result (upper bound) was matched or improved by the best competing algorithms.

In this work, we are interested in knowledge-based methods which have been successfully applied to solve many problems, including several tree-related problems [8, 24]. Specifically, we introduce the knowledge-guided iterated local search algorithm (K-ILS) for solving the PCSPG (and the RPCST). K-ILS extends our previous KTS algorithm [15] which took part in the 11th DIMACS Implementation Challenge and won several competing subcategories during the challenge. The main goal of this work aims to bring further improvements.

From an algorithmic perspective, our K-ILS algorithm belongs to the class of stochastic local search methods [27]. To be effective, K-ILS integrates dedicated and innovative ingredients for both its local optimization and perturbation phases. We identify the main contributions of the work as follows.

- First, for local optimization, K-ILS relies on an effective swap-vertex operator for tree transformations associated with an informative auxiliary evaluation function, which is shown to be extremely useful to solve a number of difficult instances with uniform or nearly uniform edge costs. For the purpose of an effective search diversification, K-ILS calls for several knowledge-guided perturbation strategies which take advantage of structure information of the given instance to direct the search towards promising areas. K-ILS additionally introduces two path-based tree transformation operators (*Connect_Customer* and *Disconnect_Customer*) to generate neighboring solutions. As such, K-ILS distinguishes itself from the existing heuristics designed for the PCSPG and RPCST problems like those presented in [1, 5, 7, 23, 31].
- Second, in terms of computational performance, K-ILS outperforms our previous KTS algorithm, which achieved an excellent performance on the RPCST and the PCSPG during the 11th DIMACS Implementation Challenge. Referring to the challenge, there are four main challenge subcategories for each problem, i.e., two *Quality Challenge* (primal bounds) subcategories regarding only the solution quality, and two *Pareto Challenge* (primal integrals) subcategories regarding both the solution quality and running time.

Experimental results (based on the same computing platform and competition rules) show that, if we have used K-ILS instead of KTS to participate in the challenge, K-ILS would have won three subcategories, while being ranked tied the first place on two subcategories, and the third place on the remaining three subcategories, with respect to other competing algorithms. Although K-ILS does not change the ranks achieved by KTS during the challenge, K-ILS does obtain better solutions than KTS on a number of difficult instances, and statistically outperforms KTS.

The reminder of the paper is organized as follows. Section 2 provides a detailed description of the proposed K-ILS algorithm. Section 3 shows computational results obtained by our algorithm. Section 4 is dedicated to an analysis of some key ingredients of the K-ILS algorithm, followed by concluding comments in the last section. A study on parameter tunings is provided in the Appendix.

2 Proposed approach

2.1 General framework

Hundreds of PCSPG and RPCST instances were collected by the 11th DIMACS Implementation Challenge, which belong to different types with very different structures. According to the No Free Lunch Theorem [40], there is no algorithm which performs globally the best on all instances. Therefore, the proposed K-ILS approach (Algorithm 1) adopts different search strategies to tackle different types of problem instances, in order to fully exploit their structures. K-ILS starts by first classifying the given instance into one of three types ('large', 'general', or 'particular') and then applying a suitable pre-processing procedure accordingly (lines 3-4). Then K-ILS enters into its main search procedure (lines 5-19), which, starting from a randomly constructed initial solution, iteratively alternates between a local optimization phase and a knowledge-guided perturbation phase. This process continues until the incumbent solution could not be further improved after W (parameter) consecutive rounds of perturbations followed by local optimization. Relative to the instance types ('large', 'general', and 'particular'), we develop dedicated local search procedures and perturbation strategies which try to explore structure information (knowledge) of the given instance to guide the search. In the following subsections, we first describe the basic and common techniques and then illustrate the dedicated search strategies respectively developed for the three instance types.

Note that even though the algorithm presented in this paper is developed for solving the PCSPG, it can be applied to solve the RPCST as well. For this,

it suffices to assign a high-enough prize to the chosen root to ensure that the root would always be included in any feasible solution of reasonable quality.

Algorithm 1: Framework of the Proposed K-ILS Approach

REQUIRE: Graph $G(V, E)$, Vertex prizes $V \to \mathbb{R}$, Edge costs $E \to \mathbb{R}$
RETURN: The best solution found T^{best}
$Type \leftarrow Identify\_Type(G)$ /* Identify instance type, Section 2.3 */
$G' \leftarrow Pre\_Process(G, Type)$ /* Pre-process the instance, Section 2.4 */
$T \leftarrow Init\_Solution(G')$ /* Construct an initial solution, Section 2.5 */
$T \leftarrow Local\_Optimize(T, Type)$ /* Local search according to its type */
$T^{best} \leftarrow T$ /* T^{best} records the best found solution */
$w \leftarrow 0$ /* w records the number of non-improving rounds */
while $w < W$ **do**
    /* Perturb T according to its type */
    $T \leftarrow Knowlege\_Guided\_Perturb(T, Type)$
    /* Improve T by local optimization according to its type */
    $T \leftarrow Local\_Optimize(T, Type)$
    if $f(T) < f(T^{best})$ **then**
        /* Update T^{best} and set w to zero if an improved solution is found */
        $T^{best} \leftarrow T$
        $w \leftarrow 0$
    else
        $w \leftarrow w + 1$ /* Otherwise, increase w by 1 */
    end if
end while

2.2 Solution representation

According to the definition of the PCSPG, once the vertices of the given graph to span are known, the optimal solution must be a minimum spanning tree (MST) over the spanned vertices, indicating that a given solution could be uniquely represented by its spanned vertices. For the sake of efficient local optimization, we adopt the solution representation used in [14, 16]. Given a specified root vertex (for the PCSPG, among all the spanned vertices, we always choose the vertex with the highest prize as the root vertex), we uniquely represent each feasible solution by a one-dimensional vector $T = \{t_i, i \in V\}$, where t_i denotes the parent vertex of vertex i if $i \in V_T$ (excluding the root vertex, V_T denotes the vertices set of T), or $t_i = Null$ otherwise.

2.3 Identification of the instance type

The 11th DIMACS Implementation Challenge has collected hundreds of PCSPG and RPCST instances of different types with highly diverse structures,

making it impossible for a single search strategy to perform uniformly well on all the instances. For this reason, we first classify the given graph $G(V, E)$ into one of three types: 'large', 'general', and 'particular'.

First, $G(V, E)$ is qualified as 'large' if $|V| > 6000$. This criterion is introduced to disable the pre-processing step (which computes and stores all the shortest paths between any pair of vertices, see Section 2.4), whose time and space costs become unaffordable for large graphs. The threshold value of 6000 is settled mainly depending on the allowed memory of the computing platform.

Second, for a graph $G(V, E)$ with $|V| \leq 6000$, G is qualified as 'general' or 'particular' according to the normalized edge cost deviation $\sigma \geq 0$ computed as:

$$\sigma = \frac{1}{|E|} \sum_{e \in E} \frac{|c_e - \overline{c_e}|}{\overline{c_e}}. \tag{2}$$

where $\overline{c_e} = \frac{\sum_{e \in E} c_e}{|E|}$ is the average cost of all the edges $e \in E$. Intuitively, a small (large) value of σ indicates a small (large) variation between the edge costs of the graph. Specifically, $\sigma = 0$ implies a uniform cost for all the edges (each edge has a same cost).

Thus, a graph with $|V| \leq 6000$ is called 'particular' if $\sigma \leq 0.05$ (settled according to our experience), and 'general' otherwise. As such, 'particular' graphs have uniform or nearly uniform edge costs while 'general' graphs may have very disparate edge costs.

Once the given graph is classified, it is possible to apply a suitable search strategy according to the characteristics of the graph. At first, for the 'general' instances, we usually observe that some vertices frequently appear in high-quality local optimal solutions while this rarely happens for other vertices. Such search information may be useful to guide the search towards more promising areas. Second, for the 'particular' instances with uniform or nearly uniform edge costs, we observe that feasibly deleting a low-prize vertex (with prize $p_i < \overline{c_e}$) would generally lead to an improving solution (only with very few exceptions when $\sigma > 0$). Inspired by this observation, we develop a swap-vertex move operator associated with an auxiliary evaluation function, as well as a swap-vertex based perturbation strategy, in order to enforce the opportunity of feasibly deleting low-prize vertices. Finally, for the 'large' instances, we try to avoid operations with unaffordable complexity. More details are given in the following subsections.

2.4 Pre-processing

For the instances of 'general' and 'particular' types, we first apply a pre-processing to calculate and store the cost of the shortest path between any pair of vertices. Using Dijkstra's algorithm with the aid of binary heap, these calculations can be achieved in $O(m + n \cdot \log n)$ [9]. These values are stored in a $n \times n$ table and can be fetched directly during the search process, instead of recalculating them repeatedly. For the 'large' instances, no pre-processing step is applied. Instead, whenever a shortest path is needed, it is calculated from scratch using Dijkstra's algorithm. For the sake of simplicity, other complex pre-processing techniques like the reduction test introduced in [38] are not applied.

2.5 Solution initialization

The K-ILS algorithm uses the following method to generate initial solutions independent of the instance type. Starting from a randomly chosen customer vertex i, we iteratively connect a good-enough customer (while guaranteeing that the objective value after insertion would not increase), using the shortest path between the selected customer vertex and the already connected solution. If there are more than one such customers available at an iteration, one of them is randomly chosen for connection. This process is repeated until no customer could be further connected. Then, in order to further improve the obtained solution, we use Kruskal's algorithm to re-construct a minimum spanning tree (MST) over the spanned vertices, to obtain a feasible solution that serves as the starting point of our K-ILS algorithm.

2.6 Basic move operators

Move operators for solution transformations are key elements of local optimization approaches. Herein, we introduce four basic move operators for generating neighboring trees, including two conventional vertex-based operators (*Insert_Vertex* and *Remove_Vertex*) developed in [7], and two path-based move operators (*Connect_Customer* and *Disconnect_Customer*) newly developed in this paper. In Section 2.8.1, we will introduce another new move operator (swap-vertex) specifically designed for the 'particular' instances.

As mentioned in Section 2.2, any solution of the PCSPG can be uniquely characterized by its spanned vertices set V_T. Consequently, given a solution $T = (V_T, E_T)$, if we insert a vertex $i \notin V_T$ to (respectively, remove a vertex $i \in V_T$ from) V_T, the resulting minimum spanning tree is a neighboring

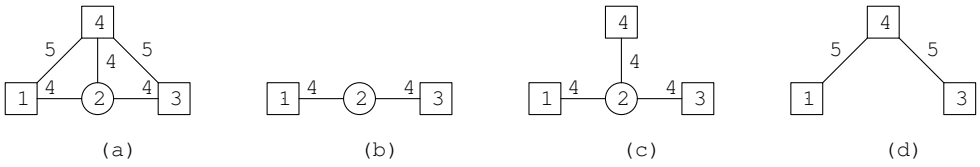

Fig. 1. An example of applying the *Insert_Vertex* and *Remove_Vertex* operators: a) input graph, b) initial solution, c) solution after inserting vertex 4, d) solution after deleting vertex 2.

solution of T (discarded if unfeasible), denoted by MST($V_T \cup \{i\}$) (respectively, MST($V_T \backslash \{i\}$). Corresponding to these two basic move operators [7], two sub-neighborhoods $N_1(T)$ and $N_2(T)$, could be defined as follows:

$$\begin{aligned} N_1(T) &= \{\text{MST}(V_T \cup \{i\}),\ \forall\ i \notin V_T\}, \\ N_2(T) &= \{\text{MST}(V_T \backslash \{i\}),\ \forall\ i \in V_T\}. \end{aligned} \tag{3}$$

Using novel dynamic data structures slightly adapted from [39] for efficiently evaluating these two sub-neighborhoods, at each iteration of local search, all the neighboring solutions belonging to $N_1(T)$ (respectively, $N_2(T)$) could be evaluated in $O(m \cdot \log\ n)$.

Fig. 1 shows an example of applying these *Insert_Vertex* and *Remove_Vertex* operators, where (a) is the original graph with three customer vertices (assume $p_1 = p_3 = p_4 = 10$, drawn in boxes) and one Steiner vertex (assume $p_2 = 0$, drawn in circle), the edge costs are drawn alongside the edges. Sub-figure (b) is an initial solution with $f(b) = 18$, while (c) and (d) are improved neighboring solutions obtained by inserting vertex 4 and removing vertex 2 subsequently, respectively corresponding to an objective value $f(c) = 12$ and $f(d) = 10$.

In addition to these two conventional move operators, we introduce two path-based move operators specifically designed (to our knowledge for the first time) for the PCSPG, which mainly focus on the customer vertices.

(1) *Connect_Customer*: Add a path to connect a customer vertex $i \notin V_T$, using the shortest path between vertex i and the incumbent solution T. The resulting neighboring solution is denoted by $T \oplus Connect\_Customer(T, i)$, whose objective value is increased by the total cost on the inserted path minus the total prize of the inserted vertices.
(2) *Disconnect_Customer*: Disconnect a customer vertex $i \in V_T$ (for convenience, we only consider leaf customer vertex here), by deleting vertex i associated with the edges which become useless (see [14] for more details about how to disconnect a leaf customer vertex). The corresponding neighboring solution is denoted by $T \oplus\ Disconnect\_Customer(T, i)$. The objective value is decreased by the total cost of the deleted edges minus

the total prize of the deleted vertices.

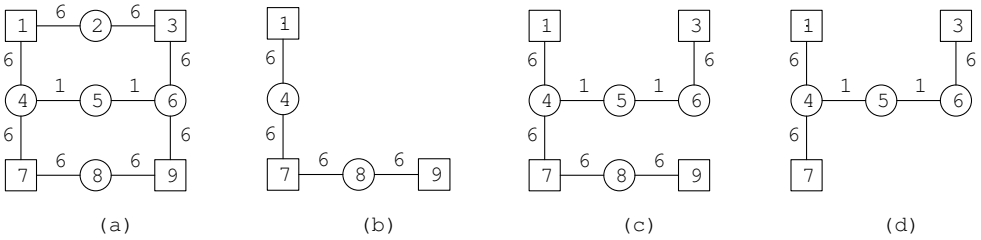

Fig. 2. An example of applying the *Connect_Customer* and *Disconnect_Customer* operators: a) input graph, b) initial solution, c) solution after connecting vertex 3, d) solution after disconnecting vertex 9.

Similarly, corresponding to these two move operators, two sub-neighborhoods $N_3(T)$ and $N_4(T)$ of the incumbent solution T are defined as follows:

$$\begin{aligned} N_3(T) &= \{T \oplus \text{Connect\_Customer}(T, i), \forall\ i \notin V_T, p_i > 0\}, \\ N_4(T) &= \{T \oplus \text{Disconnect\_Customer}(T, i), \forall \text{ leaf vertex } i \in V_T, p_i > 0\}. \end{aligned} \tag{4}$$

For example, as shown in Fig. 2, given the input graph (a) with four customer vertices (assume $p_1 = p_3 = p_7 = p_9 = 10$, drawn in boxes) and five Steiner vertices (assume $p_2 = p_4 = p_5 = p_6 = p_8 = 0$, drawn in circles), starting from a solution (b) with $f(b) = 34$, one can obtain an improved solution (c) with $f(c) = 32$ by connecting customer 3, and a further improved solution (d) with $f(d) = 30$ by disconnecting customer 9.

2.7 Search strategies for 'general' instances

To solve the instances of the 'general' type (with $|V| \leq 6000$ and $\sigma > 0.05$), we develop a tabu search (TS) procedure [22] which relies on the four move operators (*Insert_Vertex*, *Remove_Vertex*, *Connect_Customer*, *Disconnect_Customer*) defined in the last section, combined with a knowledge-guided perturbation operator relying on frequency information of each vertex to escape local optima.

2.7.1 Tabu search for local optimization

Typically, starting from a given initial solution T, our TS procedure evaluates in random order the neighboring solutions of the composed neighborhood $N(T) = N_1(T) \cup N_2(T) \cup N_3(T) \cup N_4(T)$ and iteratively accepts the first met improving solution (if no improving solution exists in $N(T)$, it accepts the

best neighboring solution in $N(T)$). To prevent local cycling, we adopt a simple tabu mechanism [22]: once a vertex is inserted into (respectively, removed from) the incumbent solution, the vertex is forbidden to be removed (respectively, inserted) again, unless the move meets the aspiration criterion, i.e., it leads to a solution better than the overall best-found solution. The search continues until the incumbent solution could not be improved after a given number M (parameter) of consecutive iterations. At this point, the best met solution T is returned as a local optimum found by tabu search. To continue the search, we turn to a perturbation phase which will modify T based on some frequency information collected during the search.

2.7.2 Frequency guided perturbations

For the 'general' instances of the PCSPG, we observe that some vertices frequently (or rarely) appear in locally optimal solutions. In order to utilize the information (knowledge) to guide the search, we employ a n-dimensional vector Z (with all values initialized to zeros) to record the frequency information of each vertex appearing in visited local optima, and use a variable Cnt (initialized to 0) to count the number of visited local optima. In order to maintain the knowledge learned during the search, after each run of tabu search, we use the obtained solution $T = (V_T, E_T)$ to update the frequency vector Z as follows: for each vertex i, if $i \in V_T$, we increase Z_i by 1; otherwise we keep Z_i unchanged. The local optima counter Cnt is also increased by 1.

With these information, we use the following strategy to perturb the incumbent solution. At first, in order to reinforce diversity, during the first ten rounds (pre-learning phase), we randomly generate initial solutions using the method described in Section 2.5, instead of perturbing the incumbent solution. After the pre-learning phase, once tabu search reaches a local optimum T, we perturb it to a new solution by flipping the status (insert an un-spanned vertex or remove a spanned vertex) of a number of vertices. Precisely, after randomly choosing a vertex i, if $i \in V_T$, we flip its status (i.e., remove i from T) with probability $\frac{Cnt-Z_i}{Cnt}$, otherwise, we flip its status (i.e., insert i into T) with probability $\frac{Z_i}{Cnt}$ (unfeasible flips are discarded directly). This action is repeated for L_{flip} (parameter) times, and then a MST is reconstructed subsequently, leading to a new solution different from T. This new solution serves as the new starting point of the next round of tabu search.

2.8 Search strategies for 'particular' instances

The above search strategies are quite effective for the instances belonging to the 'general' type, but do not perform well enough for the 'particular' instances

(with $|V| \leq 6000$ and $\sigma \leq 0.05$). Without loss of generality, in this subsection, we first introduce the techniques developed for the instances with strictly uniform edge costs ($\sigma = 0$), and then extend in Section 2.8.5 these techniques to the instances with nearly uniform edge costs ($0 < \sigma \leq 0.05$).

For convenience, we classify the vertices into two subsets, i.e., high-prize vertices with $p_i \geq \overline{c_e}$ ($\overline{c_e} = \frac{\sum_{e \in E} c_e}{|E|}$ is the average edge cost) and low-prize vertices with $p_i < \overline{c_e}$. Clearly, for the instances with strictly uniform edge costs, feasibly adding a low-prize vertex would definitively increase the objective value, leading to a worse solution. On the contrary, although feasibly deleting a low-prize vertex would always lead to an improved solution, the search process is very easy to get stuck into a local optimum where no feasible deleting move is possible. At this point, it seems very difficult to escape from the incumbent local optimum by applying the four basic move operators described in Section 2.6, even with the aid of tabu search (these observations could also be extended to the instances with nearly uniform edge costs). According to these observations, we introduce a swap-vertex based move operator (*Swap_Vertex*) associated with an auxiliary evaluation function to increase the opportunity of feasibly deleting low-prize vertices, to form an enhanced local optimization method. Furthermore, we develop several knowledge-guided perturbation operators which aim to escape from the incumbent local optimum and help the search to move to new promising areas.

2.8.1 Swap_Vertex move operator

Typically, the *Swap_Vertex* move operator exchanges a vertex $i \notin V_T$ with another vertex $j \in V_T$ and reconstructs a MST subsequently, leading to a new neighboring solution (discarded if the new solution is unfeasible). This idea is natural, but to our knowledge, it has not been well investigated in the field of Steiner tree problems, possibly due to its high computational complexity. Indeed, at each iteration there are a total of $O(|V_T|) \cdot O(|V| - |V_T|) \leq O(n^2)$ possible swap moves. If we choose to reconstruct a MST (using Kruskal's algorithm with the aid of Fibonacci heap [39]) after applying a *Swap_Vertex* move, the overall complexity would be $O(n^2) \cdot O(m + n \cdot \log n)$, being prohibitively expensive for a local search based approach.

Fortunately, for the 'particular' instances with uniform edge costs, the computation can be much simplified. Obviously, in these cases, if swapping vertex $i \notin V_T$ and vertex $j \in V_T$ leads to a feasible solution, the objective value will definitively decrease by $r_i - r_j$ (denote the objective difference by Δ hereafter), because the consumed cost remains unchanged and the collected prize increases by $r_i - r_j$. Specifically, swapping two Steiner vertices would never change the objective value. These operations can be realized in $O(n^2)$ for all the $O(n^2)$ possible swap-vertex moves.

Relative to the *Swap_Vertex* move operator, a new neighborhood $N_5(T)$ can be defined as follows.

$$N_5(T) = \{\text{MST}(V_T \cup \{i\} \backslash \{j\}),\ \forall\ i \notin V_T, j \in V_T\}. \tag{5}$$

Nevertheless, before applying any *Swap_Vertex* move, we should at first verify its feasibility. As discussed in our previous work [17], using dedicated data structures such as union-find set and leftist heap, the feasibility of all the $O(n^2)$ possible *Swap_Vertex* moves can be examined within an overall complexity of $O(m \cdot \log\ n) + O(n^2 \cdot d_{max}^2)$, where d_{max} is the maximum vertex degree of the input graph G. This complexity seems high, but remains affordable for mid-sized graphs (with $|V| \leq 6000$), especially for sparse graphs with $O(m) \approx O(n)$ (generally corresponding to a small value of d_{max}).

2.8.2 Auxiliary evaluation function

When one applies the *Swap_Vertex* operator, there are usually a large number of possible moves with the same Δ value (e.g., swapping any two Steiner vertices would lead to a $\Delta = 0$). In this case, using the objective value alone as the evaluation function will not be able to guide the search since it cannot distinguish neighboring solutions of the same objective value, even if they can lead to different search trajectories and thus different local optima. To address this problem, relative to each feasible solution T, we define as follows an auxiliary evaluation function.

Definition. Given the incumbent solution T with vertex set V_T, for each vertex $i \in V_T$, its *special degree* sd_i is defined as the number of vertices belonging to V_T which are directly reachable from i (we say a vertex $j \neq i$ is directly reachable from vertex i if edge $(i, j) \in E$), i.e.,:

$$sd_i = \sum_{j \in V_T, j \neq i, (i,j) \in E} 1. \tag{6}$$

According to this definition, if $sd_i = 1$, vertex i is directly reachable from only one vertex $j \in V_T$. Consequently, if vertex j is a low-prize vertex with $p_j < \overline{c_e}$, it implies that deleting vertex j definitively leads to an unfeasible solution. In other words, it is impossible to improve the incumbent solution by feasibly deleting low-prize vertex j. In this case, we call vertex i as a key vertex, and let $kv(T)$ denote the total number of key vertices of T.

Intuitively, the lower the value of $kv(T)$, the larger the opportunity to feasibly delete a low-prize vertex (and thus improving the solution). Inspired by this idea, during the search process, we use the objective value $f(T)$ (Eq.

(1)) as the main evaluation criterion, while adopting $kv(T)$ as an auxiliary evaluation criterion. Precisely, we say $T1$ is an improving solution over $T2$ if $f(T1) < f(T2)$ or $f(T1) = f(T2), kv(T1) < kv(T2)$, so as to distinguish the neighboring solutions with the same objective value.

2.8.3 Local optimization method

For the 'particular' instances, we combine the four basic move operators (*Insert_Vertex*, *Remove_Vertex*, *Connect_Customer*, *Disconnect_Customer*, see Section 2.6) with the *Swap_Vertex* operator (Section 2.8.1) to form an enhanced local optimization procedure. At each iteration, the search process first examines in random order (to adopt some randomness) the solutions in the combined neighborhood $N_1(T) \cup N_2(T) \cup N_3(T) \cup N_4(T)$ (generated by the four basic move operators) and accepts the first met improving neighboring solution with a lower objective value. The choice of the first met improvement strategy rather than the best improvement aims to provide the search with a higher capacity of diversification. Indeed, being less greedy, the local search procedure with the first improvement strategy is less susceptible to get trapped in local optima. If no improving solution is found in this neighborhood, it then examines in random order the solutions of neighborhood $N_5(T)$ (generated by the *Swap_Vertex* operator) and again accepts the first met improving solution (identified by the criterion described in Section 2.8.2). The optimization process iterates between the combined neighborhood and *Swap_Vertex* based neighborhood $N_5(T)$ until no improving solution can be found.

For example, Fig. 3 shows a local optimization process by combining the *Swap_Vertex* operator with the basic move operators, where sub-figure (a) is the input graph with uniform edge costs (assume $c_e = 1, \forall e \in E$, not drawn in the figure), including three high-prize vertices (assume $p_1 = p_3 = p_5 = 10$, drawn in boxes) and three low-prize vertices (assume $p_2 = p_4 = p_6 = 0$, drawn in circles). Sub-figure (b) is an initial solution with $f(b) = 4$ and $kv(b) = 2$ (both vertices 1 and 3 are key vertices). Notice that solution (b) cannot be further improved by applying the four basic move operators. However, if we swap vertex 2 and vertex 6 to get a new solution (c), although the objective value does not change, the number of key vertices decreases to 1 (only vertex 3 is a key vertex now). From this solution, we can feasibly delete low-prize vertex 4, to get an improved solution (d) with $f(d) = 3$.

2.8.4 Solution perturbation

As explained above, for the 'particular' instances, feasibly deleting a low-prize vertex leads to an improving solution. Guided by this information, once a local optimum T is reached by the local optimization procedure, we perturb

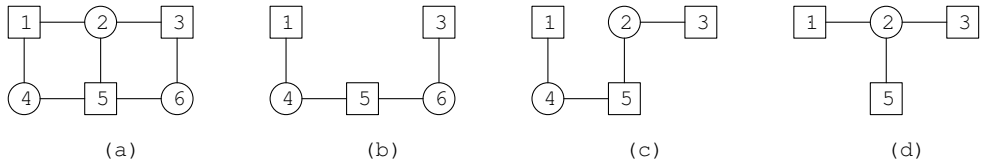

Fig. 3. An example of combining the swap-vertex move with the basic moves: a) input graph, b) initial solution, c) solution after swapping vertex 2 and vertex 6, d) solution after deleting vertex 4.

T to generate a new solution, in order to increase the opportunity of feasibly deleting a low-prize vertex (i.e., obtaining a better solution). For this purpose, we first remove from T the key vertices with a prize no greater than $\overline{c_e}$, and then insert into T the vertices with a prize no lower than $\overline{c_e}$ which are directly reachable from at least two vertices of T. This step aims to reduce the number of key vertices without increasing the objective value, thus increasing the opportunity of feasibly deleting a low-prize vertex. Furthermore, in order to be able to jump out of the current search area, we randomly swap L_{swap} (parameter) pair of low-prize vertices (discarded if unfeasible), to reach a new solution without significantly changing the objective value (specifically, swapping any two Steiner vertices would never change the objective value). Using this method, we generate a new starting solution which is fed to the local optimization method described in Section 2.8.3 for further improvement.

2.8.5 Techniques for instances with nearly uniform edge costs

Now we discuss how the above techniques can be extended to the case with varied edge costs ($\sigma > 0$). The main difficulty here is that the Δ value after swapping vertices i and j no longer strictly equals $r_i - r_j$ (the total consumed cost may vary). If we choose to calculate Δ exactly after swapping each pair of vertices, the complexity would become unaffordable again. Fortunately, we do not need to do so if σ is small enough (e.g., $\sigma \leq 0.05$), because in this case it is highly likely (only with few exceptions) that feasibly deleting (respectively, inserting) a low-prize vertex would lead to an improved (respectively, worse) solution, implying that we can still adopt the local optimization method described in Section 2.8.3 for solution improvement (regardless of the difference of edge cost while evaluating the neighboring solutions of $N_5(T)$), with the aid of the auxiliary evaluation function of Section 2.8.2 to increase the opportunity of feasibly deleting a low-prize vertex.

On the other hand, for the instances with $0 < \sigma \leq 0.05$, relative to the varied edge costs, we develop as follows a new perturbation operator. The first step (remove and insert some vertices) is kept. For the second step, we no longer randomly swap low-prize vertices. Instead, we examine all the feasible *Swap_Vertex* moves and iteratively accept the first met improving neighboring solution with $\Delta > 0$ (herein the Δ values of all the feasible *Swap_Vertex* moves

are now exactly calculated) until no improving solution exists in the whole neighborhood. The solution after perturbation serves as the new starting point of the next round of local optimization.

2.9 Search strategies for 'large' instances

For the large instances (with more than 6000 vertices), it is unaffordable to apply an operation with too high complexity. For these instances, in order to be able to obtain a feasible solution within a reasonable time, we simplify the algorithm outlined in Algorithm 1 as follows.

First, as mentioned in Section 2.4, we disable the pre-processing step, since we can not afford its high processing cost.

Second, for solution initialization, we follow the method described in Section 2.5. The only difference is that the shortest paths are temporarily calculated whenever needed, instead of pre-calculated by a pre-processing step.

Third, we adopt a simple procedure for local optimization. Given an initial solution T, the local search procedure examines in random order the neighboring solutions of its neighborhood $N(T)$ (for instances with uniform edge costs, $N(T) = N_1(T) \cup N_2(T) \cup N_5(T)$; for other instances, $N(T) = N_1(T) \cup N_2(T)$), and iteratively accepts the first met improving solution (with a lower objective value), until no improving solution exists in the whole neighborhood. Note that for the large instances, the *Connect_Customer* and *Disconnect_Customer* move operators are disabled, since without the pre-calculated shortest paths it is too expensive to apply these two move operators.

Finally, for the sake of simplicity, we no longer adopt any knowledge-guided perturbation strategy. Instead, whenever a local optimum is reached, we reconstruct from scratch a new solution (using the solution initialization method mentioned above) which serves as the starting point of the next round of local optimization method.

2.10 Differences between K-ILS and KTS

As indicated above, K-ILS is based on our previous KTS algorithm [15], by following the same iterated local search framework and sharing many important basic ideas. On the other hand, K-ILS adopts some refined techniques for local optimization and solution perturbation. The differences for handling the 'general' instances and the 'particular' instances are summarized as follows (there is no big difference between K-ILS and KTS for the 'large' instances).

For the 'general' instances, K-ILS is much simplified compared to KTS. During the local search phase, K-ILS uses a simple rule to prevent local cycling, instead of the complex tabu mechanism of KTS which relies on several tabu tenures (parameters). During the perturbation phase, K-ILS only adopts a simple knowledge-guided perturbation strategy, instead of the two (weak and strong) strategies used by KTS.

For the 'particular' instances, K-ILS adopts several different search mechanisms compared to KTS. At first, based on the concept of "special degree" of each vertex defined in [14], K-ILS newly defines the concept of "key vertex", and modifies accordingly the auxiliary evaluation criterion for identification of improving solutions. Furthermore, K-ILS re-designs the perturbation strategies, being very different from those used by KTS. Actually, experimental results (detailed in the next section) show that, compared to KTS, K-ILS is able to yield (within the same allowed time, tested on the same platform) many better results on the 'particular' instances, indicating the contribution of these new techniques to the performance of the proposed algorithm.

3 Experimental results

To assess the proposed K-ILS algorithm for solving the PCSPG and RPCST, we evaluate our algorithm on the benchmark instances used by the 11th DIMACS Implementation Challenge [29] and carry out our experiments under the same computing platform and competition rules of the challenge, which are summarized as follows.

First, for each competing algorithm, a result obtained on each benchmark instance is measured under two tracks: Quality Challenge and Pareto Challenge.

(1) The *Quality Challenge* (primal bound) track considers only solution quality (the best solution found within the allowed time), regardless of the actual time to attain the best solution.
(2) The *Pareto Challenge* (primal integral) track considers both solution quality and computing time needed to attain the best solution, as precisely defined in [4]. Generally, a lower value of primal integral indicates a better performance in terms of Pareto Challenge.

Furthermore, for each challenge track, the overall performance of a given competing algorithm on a set of benchmark instances, with respect to other competing algorithms, is measured under two subcategories, each being aggregated into a single score.

(1) *Formula 1 (points-based method)*: For each benchmark instance, all the

competing algorithms are ranked according to their results (primal bounds or primal integrals, respectively). Then the best algorithm gets 10 points, the second gets 6 points, then 4, 3, 2, 1 points. The algorithms that do not produce results or are not among the top 6 ones do not get any point. Finally, for each competing algorithm, the points on all the benchmark instances are collected together as the final score. The higher the obtained points, the better the overall performance of a competing algorithm, with respect to other competing algorithms.

(2) *Average (geometric mean)*: For each competing algorithm, the results obtained on different instances are aggregated to a score by simply computing their geometric mean (in terms of primal bounds or primal integrals, respectively). From this point of view, a lower geometric mean value indicates a better overall performance of a competing algorithm.

Therefore, for each of the seven problem variants (including both the PCPSG and the RPCST) of the 11th DIMACS Challenge, all the competing algorithms were uniformly measured and ranked under four main challenge subcategories, as detailed in the next subsections.

To evaluate the performance of our K-ILS algorithm, we include the detailed results from all the competing algorithms (including our previous KTS algorithm [15]), together with the results achieved by the K-ILS algorithm. To make the assessment as meaningful as possible, the K-ILS algorithm was run on the same computing platform used for the challenge under the competition rules[2]. Precisely, K-ILS was executed on a cluster with 32 compute nodes (one job per node), each node with an Intel Xeon X5672 3.20 GHz processor (8 cores) and 48 GB RAM. To solve a given instance, K-ILS was executed repeatedly and independently (each independent run using the stopping criterion as shown in Algorithm 1) until a total time of two CPU hours was reached (this is the time limit used by the challenge). It is worth noting that according to the competition rules, the number of restarts of each algorithm during the 2 hours run and the best objective value of each restart were not recorded. However, one can get an idea about this information from Tables 6 – 8 (columns "Restats" and "*SD*") of Section 4.

[2] We are grateful to Dr. Gerald Gamrath, member of the 11th DIMACS Challenge organization team for his kind help to run the K-ILS code on the computing platform of the challenge and collect the results. The executable code of our K-ILS algorithm as well as the detailed results are available online at `http://www.info.univ-angers.fr/pub/hao/pcspg.html`

3.1 Parameter Setting

As described in Section 2, K-ILS requires four independent parameters: L_{flip} and L_{swap} used to control the perturbation strength respectively for the 'general' and 'particular' instances, M and W used in the stopping conditions of the tabu search procedure (for local optimization) and each independent run of the K-ILS algorithm. Given that the test instances have very different structures, it is extremely difficult to obtain a set of parameters which perform uniformly the best on all the instances. Following our experiments (see Appendix A for more details), we observe that the values listed in Table 1 perform reasonably well on the test instances, thus we adopt them as the default parameter setting for our experiments.

Table 1
Default setting of each parameter.

Parameter	Description	Default Setting
L_{flip}	Perturbation strength for the general instances, Section 2.7.2	n
L_{swap}	Perturbation strength for the particular instances, Section 2.8.4	$0.5n$
M	Stopping condition of tabu search, Section 2.7.1	30
W	Stopping condition of each independent run of K-ILS, Section 2.1	50

3.2 Results on the RPCST

As explained in Section 2.1, although our K-ILS algorithm is designed for the PCSPG, it can be used to solve the RPCST as well by simply assigning to the chosen root a high enough prize. Herein, we first present the challenge results on the 29 RPCST benchmark instances adopted by the 11th DIMACS Challenge (according to the criterion described in Section 2.3, all these 29 instances are 'general' instances). The results of the competing algorithms (obtained within two CPU hours for each instance) on these instances are given in Table 2 and Table 3, respectively in terms of *Quality Challenge* and *Pareto Challenge*. In each table, the first column indicates the instance name, while the following ten columns respectively indicate the results of each competing algorithm. The last column shows the results of our K-ILS algorithm. Additionally, the last three rows summarize the overall performance of each algorithm. According to the challenge rules, corresponding to both *Quality Challenge* and *Pareto Challenge*, the competing algorithms are ranked under two different subcategories, in terms of *Formula 1* (points based method) and *Average* (geometric mean). Accordingly, row *Formula 1 (Original)* gives the original point of each competing algorithm released during the 11th DIMACS Challenge, row *Formula 1 (New)* gives the new point of each competing algorithm if K-ILS instead of KTS is used to participate in the Challenge, and row *Average* shows the geometric mean of the results obtained by each algorithm. As explained before, a higher value of *Formula 1* or a lower value of *Average* indicates a better overall performance of a competing algorithm.

Table 2. Challenge results in terms of 'Quality' criterion (primal bounds) on 29 RPCST instances (within two CPU hours).

Instance	AB	heinz-mc	heinz-no-dc	heinz-no-pre	mozartballs	polito	schimidt	scipjack	scipjackspx	KTS	K-ILS
i101M1	**109271.5**	109271.5	**109271.5**	**109271.5**	**109271.5**	**109271.5**	**109271.5**	**109271.5**	**109271.5**	**109271.5**	**109271.5**
i101M2	387041.0	332485.1	317573.6	320680.3	**315925.3**	317865.3	341795.5	**315925.3**	**315925.3**	**315925.3**	**315925.3**
i101M3	622113.3	448429.9	367412.7	368053.1	**355625.4**	359306.8	385703.6	**355625.4**	**355625.4**	**355625.4**	**355625.4**
i102M1	**104065.8**	104065.8	**104065.8**	**104065.8**	**104065.8**	**104065.8**	**104065.8**	**104065.8**	**104065.8**	**104065.8**	**104065.8**
i102M2	354337.2	363517.0	354053.8	355152.9	**352538.8**	354874.1	363303.1	**352538.8**	**352538.8**	**352538.8**	**352538.8**
i102M3	580556.3	533274.3	475309.1	467402.0	**454365.9**	459372.7	506798.4	**454365.9**	**454365.9**	**454365.9**	**454365.9**
i103M1	**139749.4**	139749.4	**139749.4**	**139749.4**	**139749.4**	**139749.4**	**139749.4**	**139749.4**	**139749.4**	**139749.4**	**139749.4**
i103M2	408876.2	425641.0	408561.2	409524.2	**407834.2**	411325.8	414413.9	**407834.2**	**407834.2**	**407834.2**	**407834.2**
i103M3	606110.0	567154.9	462642.5	462207.2	**456125.5**	460342.6	510162.8	**456125.5**	**456125.5**	**456125.5**	**456125.5**
i104M2	90900.2	**89920.8**	**89920.8**	**89920.8**	**89920.8**	**89920.8**	**89920.8**	**89920.8**	**89920.8**	**89920.8**	**89920.8**
i104M3	97149.8	99087.4	**97148.8**	97150.8	**97148.8**	**97148.8**	**97148.8**	**97148.8**	**97148.8**	**97148.8**	**97148.8**
i105M1	**26717.2**	26717.2	**26717.2**	**26717.2**	**26717.2**	**26717.2**	**26717.2**	**26717.2**	**26717.2**	**26717.2**	**26717.2**
i105M2	104496.5	100742.6	**100269.6**	**100269.6**	**100269.6**	101793.6	**100269.6**	**100269.6**	**100269.6**	**100269.6**	**100269.6**
i105M3	110414.2	114535.9	111592.2	111544.2	**110351.2**	110616.2	114983.2	**110351.2**	**110351.2**	**110351.2**	**110351.2**
i201M2	359774.2	359774.2	355844.0	355825.5	**355467.7**	356008.1	**355467.7**	**355467.7**	**355467.7**	**355467.7**	**355467.7**
i201M3	724259.7	747658.1	667373.4	664638.8	**628833.6**	656766.0	634950.9	**628833.6**	**628833.6**	**628833.6**	**628833.6**
i201M4	941281.8	1078178.2	795718.9	883643.1	**773398.3**	790722.4	819724.7	**773398.3**	**773398.3**	**773398.3**	**773398.3**
i202M2	291469.8	356354.2	291645.1	290768.4	**288946.8**	304695.9	**288946.8**	**288946.8**	**288946.8**	**288946.8**	**288946.8**
i202M3	1207417.6	589577.6	455150.5	451039.1	**419184.2**	436537.5	430188.0	**419184.2**	**419184.2**	**419184.2**	**419184.2**
i202M4	2033361.1	736072.6	489015.2	504721.2	**430034.3**	437662.3	489456.1	**430034.3**	**430034.3**	**430034.3**	**430034.3**
i203M2	1055699.2	565329.6	461078.7	468648.4	**459894.8**	515809.9	**459894.8**	**459894.8**	**459894.8**	**459894.8**	**459894.8**
i203M3	1817008.9	931922.4	660988.8	655975.2	**643062.0**	663513.4	666414.9	**643062.0**	**643062.0**	**643062.0**	**643062.0**
i203M4	1198975.1	1239234.0	707381.6	802540.0	**677733.1**	689472.2	707384.7	**677733.1**	**677733.1**	**677733.1**	**677733.1**
i204M2	**161700.5**	161700.5	**161700.5**	**161700.5**	**161700.5**	**161700.5**	**161700.5**	**161700.5**	**161700.5**	**161700.5**	**161700.5**
i204M3	377301.3	319133.5	258833.5	293949.0	**245287.2**	255076.8	344623.3	**245287.2**	**245287.2**	**245287.2**	**245287.2**
i204M4	754676.8	319413.6	412682.1	300827.5	**245287.2**	251987.6	344623.3	**245287.2**	**245287.2**	**245287.2**	**245287.2**
i205M2	610712.0	620194.2	586888.9	595942.8	**571031.4**	612473.0	571459.1	**571031.4**	**571031.4**	**571031.4**	**571031.4**
i205M3	976501.8	1030486.9	703489.3	1003969.0	**672403.1**	680037.4	778670.3	**672403.1**	**672403.1**	**672403.1**	**672403.1**
i205M4	1004104.9	1423986.6	984497.1	781172.5	**713973.6**	725387.4	844022.1	**713973.6**	**713973.6**	**713973.6**	**713973.6**
Formula 1 (Original)	18.4	2.9	39.8	29.0	**143.3**	45.0	45.8	**143.3**	**143.3**	**143.3**	-
Formula 1 (New)	18.4	2.9	39.8	29.0	**143.3**	45.0	45.8	**143.3**	**143.3**	-	**143.3**
Average	511417.0	430218.8	362312.7	365444.1	**339422.7**	347264.3	367760.1	**339422.7**	**339422.7**	**339422.7**	**339422.7**

Table 3. Challenge results in terms of 'Pareto' criterion (primal integrals) on 29 RPCST instances (within two CPU hours).

Instance	AB	heinz-mc	heinz-no-dc	heinz-no-pre	mozartballs	polito	schimidt	scipjack	scipjackspx	KTS	K-ILS
i101M1	0.0	7200.0	0.4	0.4	1.0	0.0	0.0	0.4	0.4	0.0	0.0
i101M2	1323.0	435.4	43.9	113.1	0.1	44.2	545.0	0.4	0.4	0.0	0.0
i101M3	3084.4	1512.8	257.2	292.5	0.1	81.3	561.5	0.4	0.4	0.1	0.1
i102M1	0.0	7200.0	0.4	0.4	1.0	0.0	0.0	0.4	0.4	0.0	0.0
i102M2	36.6	223.4	39.4	71.7	0.1	47.5	213.4	0.4	0.2	0.0	0.1
i102M3	1565.2	1069.4	322.5	207.5	0.2	162.9	744.9	0.4	0.4	0.0	0.0
i103M1	0.0	7200.0	0.5	0.4	1.1	0.0	0.0	0.4	0.4	0.0	0.0
i103M2	18.4	326.5	16.8	32.8	0.1	61.5	114.3	0.4	0.4	0.0	0.1
i103M3	1782.6	1414.3	105.9	104.0	0.2	69.9	762.7	0.4	0.4	0.0	0.0
i104M2	77.6	0.1	0.4	0.4	0.1	0.0	0.0	0.4	0.4	0.0	0.0
i104M3	0.2	179.2	4.0	11.3	0.3	0.1	0.0	0.4	0.4	0.0	0.0
i105M1	0.0	7200.0	0.4	0.4	1.1	0.0	0.0	0.4	0.4	0.0	0.0
i105M2	291.2	43.4	4.6	4.5	0.1	107.9	0.0	0.4	0.4	0.0	0.0
i105M3	4.2	341.5	84.8	108.8	0.3	28.0	290.1	0.4	0.4	0.0	0.0
i201M2	86.2	7200.0	9.7	9.9	0.2	11.1	0.1	1.4	1.4	0.1	0.0
i201M3	949.0	1213.7	452.8	417.5	0.4	307.8	69.5	1.5	1.6	0.1	0.1
i201M4	1284.3	2042.4	269.5	1128.9	0.3	199.2	407.0	1.4	1.4	0.1	0.1
i202M2	62.3	1415.6	83.5	68.9	0.2	372.3	0.1	1.5	1.4	0.1	0.0
i202M3	4700.4	2084.5	602.6	583.5	0.2	286.7	184.3	1.4	1.5	0.1	0.1
i202M4	5677.3	3031.9	931.2	1200.5	0.2	173.1	874.2	1.5	1.4	0.1	0.1
i203M2	4063.5	1443.0	40.4	191.2	0.2	780.6	0.1	1.5	1.5	0.1	0.0
i203M3	4651.8	2356.2	223.6	173.2	0.4	227.1	252.4	1.5	1.7	0.1	0.1
i203M4	3130.1	3328.1	352.3	1154.4	0.3	155.3	301.9	1.4	1.6	0.1	0.1
i204M2	0.0	7200.0	0.9	0.9	0.2	0.4	0.1	2.0	2.2	0.1	0.1
i204M3	2519.2	1705.7	1002.5	1204.5	0.2	276.6	2075.4	2.2	2.1	0.1	0.1
i204M4	4859.8	1694.8	3043.2	1432.5	0.2	206.5	2075.4	2.1	2.0	0.1	0.0
i205M2	467.8	7200.0	197.7	303.9	0.2	487.3	5.5	1.4	1.4	0.1	0.1
i205M3	2242.2	2642.3	373.7	2387.9	0.3	81.9	982.7	1.6	1.4	0.1	0.1
i205M4	2080.4	3633.6	1996.6	716.6	1.0	116.8	1109.5	1.5	1.6	0.1	0.1
Formula 1 (Original)	59.0	2.0	14.3	13.0	132.0	51.3	67.3	77.8	75.8	**261.3**	-
Formula 1 (New)	59.0	2.0	14.3	13.0	132.0	51.3	67.3	77.8	75.8	-	**261.3**
Average	180.8	1288.8	55.7	72.3	0.3	40.4	38.6	0.9	0.9	**0.0**	**0.0**

On the one hand, regarding the *Quality Challenge* criterion (primal bounds), Table 2 shows that, among the ten competing algorithms, four algorithms (mozartballs [13], scipjack and scipjackspx [18], as well as our KTS algorithm [15]), perform tied the best on these 29 RPCST instances (with all instances solved to optimality). Therefore, KTS was ranked tied the first place on both subcategories *Formula 1 (Original)* and *Average*. Moreover, if K-ILS instead of KTS is used for the Challenge, K-ILS could also reach the optimal results on all these 29 instances (within two CPU hours for each instance), leading to the same ranks on both subcategories *Formula 1 (New)* and *Average*.

Furthermore, we used the Wilcoxon test to check the statistical differences between the results of K-ILS and each competing algorithm, which respectively leads to a p-value of 9.63×10^{-7} (K-ILS vs. AB), 1.62×10^{-6} (K-ILS vs. heinz-mc), 4.59×10^{-6} (K-ILS vs. heinz-no-dc), 2.72×10^{-6} (K-ILS vs. heinz-no-pre), 1.00 (K-ILS vs. morzatball), 2.73×10^{-6} (K-ILS vs. polito), 2.21×10^{-5} (K-ILS vs. schimidt), 1.00 (K-ILS vs. scipjack), 1.00 (K-ILS vs. scipjackspx), and 1.00 (K-ILS vs. KTS). It indicates, in terms of *Quality Challenge* K-ILS performs statistically better than six competing algorithms (AB, heinz-mc, heinz-no-dc, heinz-no-pre, polito and schimidt), while obtaining the same results with respect to the remaining four algorithms (morzatball, scipjack, scipjackspx and KTS).

On the other hand, regarding the *Pareto Challenge* criterion (primal integrals) which concerns both the solution quality and running time, we observe from Table 3 that KTS clearly dominates other competing algorithms on both subcategories *Formula 1 (Original)* and *Average*. Furthermore, if we use K-ILS instead of KTS to participate in the Challenge, K-ILS would also be ranked the first place on both subcategories *Formula 1 (New)* and *Average*, implying an excellence performance of K-ILS on these 29 benchmark instances.

Similarly, the Wilcoxon test in terms of *Pareto Challenge* respectively reveals a p-value of 4.23×10^{-6} (K-ILS vs. AB), 7.24×10^{-8} (K-ILS vs. heinz-mc), 7.24×10^{-8} (K-ILS vs. heinz-no-dc), 7.24×10^{-8} (K-ILS vs. heinz-no-pre), 7.24×10^{-7} (K-ILS vs. morzatball), 9.63×10^{-7} (K-ILS vs. polito), 4.59×10^{-6} (K-ILS vs. schimidt), 7.24×10^{-8} (K-ILS vs. scipjack), 7.24×10^{-8} (K-ILS vs. scipjackspx), and 4.14×10^{-1} (K-ILS vs. KTS), indicating that in terms of *Pareto Challenge*, K-ILS performs similarly with KTS, while clearly outperforms all the remaining algorithms.

To conclude, the comparisons on the RPCST clearly indicate the effectiveness and efficiency of the proposed K-ILS algorithm with respect to other competing algorithms, although it does not performs very differently from our previous KTS algorithm.

3.3 Results on the PCSPG

For the PCSPG, hundreds of instances (classified into eight groups) were collected by the 11th DIMACS Implementation Challenge, among which 32 most challenging instances (four instances per group) were finally adopted as benchmarks to evaluate the competing algorithms. Like in Section 3.2, we show in Table 4 and Table 5 the results of these 32 instances obtained by the 12 competing algorithms as well as our new K-ILS algorithm, respectively in terms of *Quality Challenge* and *Pareto Challenge*. Note that on all these 32 instances, the previous best known primal bounds existing in the literature (published before the challenge, if applicable) were easily matched or improved by the best competing algorithms, indicating that the 11th DIMACS Challenge definitively extended the research on the PCSPG.

Additionally, according to the criterion described in Section 2.3 for instance types, these 32 PCSPG instances include 14 'general' instances, 12 'particular' instances and 6 'large' instances. In Table 4 and Table 5, the first 14 instances are 'general' instances, while the following 12 ones are 'particular' instances, and the final 6 ones are 'large' instances.

On the one hand, as shown in Table 4, in terms of *Quality Challenge*, our KTS algorithm was ranked the third place on both subcategories *Formula 1* (Original) and *Average*, performing slightly worse than algorithms "mozartballs" and "staynerd" [13], and clearly better than other competing algorithms. Compared to our previous KTS algorithm, K-ILS is able to find respectively 11 better, 18 equal and 3 worse results. Especially, on the 12 'particular' instances which are known to be extremely difficult for all the competing algorithms, K-ILS obtains 7 better results and only 1 worse result compared to KTS. Furthermore, if we use K-ILS instead of KTS to participate in the challenge, the score in terms of *Formula 1* (New) would increase from 142.8 to 155.5, and the score in terms of *Average* would decrease from 5129.0 to 5125.1. Although K-ILS is still ranked the third place on both challenge subcategories, the gaps between K-ILS and "mozartballs" (as well as "staynerd") are much reduced, indicating that K-ILS is an improved version over KTS in terms of *Quality Challenge*, especially on the 'particular' instances with uniform or nearly uniform edge costs.

As for the RPCST, we used the Wilcoxon test to check the statistical significances of the differences in terms of *Quality Challenge*, which respectively reveals a p-value of 9.67×10^{-4} (K-ILS vs. AB), 3.19×10^{-7} (K-ILS vs. heinz-dc), 4.32×10^{-8} (K-ILS vs. heinz-mc), 3.18×10^{-7} (K-ILS vs. heinz-no-dc), 2.58×10^{-8} (K-ILS vs. heinz-no-pre), 4.39×10^{-1} (K-ILS vs. morzatball), 2.78×10^{-2} (K-ILS vs. polito), 5.34×10^{-7} (K-ILS vs. schimidt), 1.62×10^{-4} (K-ILS vs. scipjack), 5.70×10^{-5} (K-ILS vs. scipjackspx), 8.08×10^{-1} (K-ILS

Table 4. Challenge results in terms of the 'Quality' criterion (primal bounds) on 32 PCSPG instances (within two CPU hours).

Instance	AB	heinz-dc	heinz-mc	heinz-no-dc	heinz-no-pre	mozartballs	polito	schimidt	scipjack	scipjackspx	staynerd	KTS	K-ILS
C13-A	**236**	265	398	267	439	**236**	237	245	**236**	**236**	**236**	**236**	**236**
C19-B	**146**	180	1015	180	6566	**146**	**146**	157	**146**	**146**	**146**	**146**	**146**
D03-B	**1509**	1611	2009	1597	8514	**1509**	**1509**	1579	**1509**	**1509**	**1509**	**1509**	**1509**
D20-A	**536**	592	2250	592	2490	**536**	**536**	542	**536**	**536**	**536**	**536**	**536**
P400-3	**2951725**	3114184	4449851	3087932	5125476	**2951725**	**2951725**	3010150	**2951725**	**2951725**	**2951725**	**2951725**	**2951725**
P400-4	**2852956**	3094213	4450178	3067298	4962336	**2852956**	**2852956**	2942801	**2852956**	**2852956**	**2852956**	**2852956**	**2852956**
K400-7	511666	477785	499183	481221	543106	**474466**	523885	476299	**474466**	**474466**	**474466**	**474466**	**474466**
K400-10	408913	395859	409149	397358	448964	**394191**	403804	406658	**394191**	**394191**	**394191**	**394191**	**394191**
i640-001	**2932**	**2932**	**2932**	**2932**	**2932**	**2932**	3066	3414	**2932**	**2932**	**2932**	**2932**	**2932**
i640-221	29356	11766	14573	11766	29356	**8400**	8603	11630	28108	28108	8427	**8400**	**8400**
i640-321	105188	41683	45616	41683	106360	28799	28797	41235	105092	105092	28809	**28787**	**28787**
i640-341	29893	42000	45429	41879	95656	29750	29692	41990	29734	29867	29732	**29666**	29671
a2000RandGraph-2	**1483.8**	1642.4	1927.5	1642.4	1937.7	**1483.8**	1484.2	1535.7	**1483.8**	**1483.8**	**1483.8**	1487.0	1485.8
a4000RandGraph-3	**3406.6**	3862.5	5762.0	3862.5	6105.4	**3406.6**	3407.5	3484.8	6102.4	6102.4	**3406.6**	3410.2	3412.0
hc10p	60150	71220	72801	71220	77425	60016	**59738**	73804	66741	68742	59981	59816	59787
hc11u	1439	1317	1407	1317	1555	1116	1117	1297	1152	1157	1116	1117	**1115**
hc12p	308227	280940	293216	280940	308227	236254	**234977**	293165	308027	308027	235958	236568	236380
hc12u	3083	2618	2825	2618	3083	2223	2224	2552	3081	3081	2223	2221	**2217**
bip52nu	**222**	280	288	283	302	**222**	223	289	225	223	223	223	**222**
bip62nu	**214**	257	277	257	302	**214**	**214**	254	217	217	**214**	**214**	**214**
cc3-12nu	100	104	111	104	114	**95**	96	108	98	99	**95**	**95**	**95**
cc12-2nu	575	616	693	616	699	577	567	658	570	575	571	567	**565**
drosophila001	8310.9	8286.4	8286.4	8286.4	8304.3	**8274.0**	8288.3	8286.4	8286.4	8286.4	**8274.0**	8278.5	8275.5
HCMV	**7371.5**	7376.2	7375.5	7375.4	7385.3	**7371.5**	7378.2	7376.2	**7371.5**	**7371.5**	**7371.5**	**7371.5**	**7371.5**
lymphoma	**3341.9**	3375.5	3393.4	3373.7	3419.3	**3341.9**	3349.1	3377.4	**3341.9**	**3341.9**	**3341.9**	**3341.9**	**3341.9**
metabol-expr-mice-1	11359.1	11409.5	11491.5	11438.5	11903.6	**11346.9**	11901.9	11407.3	11832.6	11832.6	**11346.9**	11347.1	11349.2
a8000RandGraph-1.2	4796.5	-	4790.3	-	-	**4720.0**	4720.5	4789.8	4790.8	4790.8	**4720.0**	4756.6	4756.1
a14000RandGraph-1.5	10514.2	10412.5	10512.3	10351.6	10515.3	**9475.6**	9475.7	9867.3	10513.8	10513.8	**9475.6**	9577.0	9576.0
handsd04	779.1	585.1	782.1	585.1	792.9	494.4	587.7	496.5	791.2	791.2	**493.8**	741.0	734.7
handbd13	**13.2**	-	-	-	-	**13.2**	**13.2**	**13.2**	**13.2**	-	**13.2**	**13.2**	**13.2**
handsi03	56.2	56.2	56.2	56.2	56.3	**56.1**	56.3	56.2	56.2	56.2	**56.1**	56.2	56.2
handbi07	**151.0**	-	-	-	-	**151.0**	151.1	**151.0**	**151.0**	-	**151.0**	**151.0**	**151.0**
Formula 1 (Original)	67.3	14.6	4.6	13.6	2.6	166.5	92.6	35.0	65.7	57.7	**169.0**	142.8	-
Formula 1 (New)	66.0	14.6	4.6	13.6	2.6	162.2	89.6	34.0	65.7	57.2	**166.5**	-	155.5
Average	5831.9	5965.5	7062.5	5694.7	8935.5	5058.5	5130.6	5601.8	5903.5	5913.3	**5057.4**	5129.0	5125.1

Table 5. Challenge results in terms of the 'Pareto' criterion (primal integrals) on 32 PCSPG instances (within two CPU hours).

Instance	AB	heinz-dc	heinz-mc	heinz-no-dc	heinz-no-pre	mozartballs	polito	schimidt	scipjack	scipjackspx	staynerd	KTS	K-ILS
C13-A	0.0	815.2	2937.3	851.5	7200.0	0.0	30.6	264.5	0.3	0.4	0.0	0.0	0.0
C19-B	0.0	1400.8	6169.6	1399.4	7200.0	0.2	0.8	504.5	1.0	1.0	0.2	0.0	0.0
D03-B	0.0	12446.3	1848.0	407.0	7200.0	0.0	0.1	319.2	1.0	1.0	0.0	0.0	0.1
D20-A	0.1	792.4	5492.5	786.0	7200.0	0.3	1.6	79.8	17.1	16.9	0.3	0.0	0.0
P400-3	0.0	188.0	2460.1	351.3	7200.0	0.0	0.1	139.8	0.4	0.3	0.0	0.0	0.0
P400-4	0.0	509.9	2587.0	521.7	7200.0	0.0	0.1	219.8	0.3	0.3	0.0	0.0	0.0
K400-7	523.5	364.1	366.3	107.0	7200.0	0.0	679.3	27.7	0.2	0.3	0.0	0.0	0.0
K400-10	259.2	251.8	264.1	66.1	7200.0	0.0	171.5	220.7	0.1	0.4	0.0	0.0	0.1
i640-001	0.0	943.1	0.5	3.8	69.5	0.0	314.8	1016.5	0.0	0.0	0.0	0.0	0.0
i640-221	5139.8	2433.2	3070.7	2074.0	7200.0	14.4	186.6	2000.3	5050.2	5050.2	25.8	0.4	0.4
i640-321	5229.6	2609.7	2701.1	2255.3	7200.0	7.1	18.2	2174.0	5229.5	5229.5	8.3	0.4	0.4
i640-341	54.7	2128.6	2501.2	2118.4	7200.0	22.0	9.4	2113.3	59.5	74.8	16.8	0.4	2.5
a2000RandGraph-2	0.0	729.0	1670.4	726.3	7200.0	0.2	2.8	243.3	1093.1	1094.5	0.2	15.8	11.8
a4000RandGraph-3	0.0	1942.4	3024.8	1918.7	7200.0	0.5	4.1	161.6	3199.2	3199.5	0.5	12.2	13.6
hc10p	49.4	1164.4	1300.8	1164.2	7200.0	35.8	0.6	1372.2	850.2	1010.9	31.7	9.9	7.7
hc11u	1616.3	1112.4	1503.1	1111.0	7200.0	1.0	11.1	1004.8	730.1	752.8	1.0	8.4	7.5
hc12p	1711.1	1254.9	1490.6	1248.0	7200.0	52.4	7.0	1429.2	1714.8	1714.7	48.2	61.5	57.4
hc12u	2013.1	1157.3	1601.6	1151.7	7200.0	7.5	13.0	934.0	2017.0	2017.0	7.5	8.5	18.5
bip52nu	0.0	1504.2	1662.8	1553.7	7200.0	1.6	33.1	1669.2	103.7	49.1	32.6	32.9	0.6
bip62nu	0.0	1207.1	1641.7	1206.3	7200.0	0.4	0.7	1133.9	108.0	104.9	0.3	0.2	0.2
cc3-12nu	360.0	631.7	1039.2	626.5	7200.0	0.4	77.5	866.8	221.5	291.9	0.4	0.3	0.3
cc12-2nu	100.2	594.7	1315.2	589.4	7200.0	129.2	2.3	995.8	68.5	123.2	56.4	3.9	26.8
drosophila001	32.0	7189.0	283.5	331.2	7200.0	1.1	19.3	11.3	38.7	39.9	1.0	4.1	2.1
HCMV	0.0	1776.5	50.2	52.9	7200.0	0.3	8.8	4.8	8.2	8.0	0.3	0.1	0.1
lymphoma	0.0	1677.7	117.8	78.7	7200.0	0.1	16.2	75.7	988.5	992.3	0.1	0.0	0.0
metabol-expr-mice-1	7.8	3907.9	101.7	402.1	7200.0	0.1	336.3	38.2	298.6	298.3	0.1	1.3	1.7
a8000RandGraph-1.2	114.9	7200.0	488.0	7200.0	7200.0	1.0	6.3	105.3	255.8	254.2	1.1	55.8	54.9
a14000RandGraph-1.5	711.2	771.5	1893.8	2117.1	7200.0	3.7	9.8	286.4	1181.5	1168.5	5.7	79.2	77.0
handsd04	2636.6	3346.0	2978.0	3281.7	7200.0	10.6	1159.6	40.2	2737.3	2737.2	1.4	2404.4	2376.3
handbd13	14.8	7200.0	7200.0	7200.0	7200.0	8.6	123.0	2.1	747.8	743.5	8.4	1.0	1.1
handsi03	7.9	48.2	3307.7	3709.3	7200.0	1.3	31.6	1.1	734.7	733.8	1.3	5.0	4.9
handbi07	0.9	7200.0	7200.0	7200.0	7200.0	6.9	111.7	2.2	735.5	741.1	6.7	1.1	0.8
Formula 1 (Original)	146.8	1.0	1.0	6.0	0.0	155.2	89.0	45.0	25.2	19.2	159.2	**184.4**	-
Formula 1 (New)	142.8	1.0	1.0	6.0	0.0	160.0	90.0	45.0	25.2	19.2	163.0	-	**178.8**
Average	22.0	1330.0	1107.3	771.9	6230.8	**2.2**	16.7	185.1	101.0	103.5	**2.2**	3.1	3.0

vs. staynerd), and 3.25×10^{-2} (K-ILS vs. KTS). These statistics demonstrate that there is no significant difference between K-ILS and the top two leading algorithms (morzatball and staynerd), while K-ILS clearly outperforms the remaining 10 algorithms (including our previous KTS algorithm, meaning that K-ILS is an improved version over KTS).

On the other hand, as shown in Table 5, in terms of *Pareto Challenge*, our previous KTS algorithm won on subcategory *Formula 1* (Original) and was ranked the third place on subcategory *Average* (also performing slightly worse than "mozartballs" and "staynerd"). Indeed, if we replace KTS by K-ILS to participate in the challenge, the final ranks will remain unchanged, without significant difference (the scores on subcategory *Formula 1* (New) and subcategory *Average* both decrease slightly).

Similarly, the Wilcoxon test in terms of *Pareto Challenge* respectively reveals a p-value of 6.04×10^{-3} (K-ILS vs. AB), 1.54×10^{-8} (K-ILS vs. heinz-dc), 1.54×10^{-8} (K-ILS vs. heinz-mc), 1.54×10^{-7} (K-ILS vs. heinz-no-dc), 1.54×10^{-8} (K-ILS vs. heinz-no-pre), 8.48×10^{-1} (K-ILS vs. morzatball), 1.96×10^{-2} (K-ILS vs. polito), 7.43×10^{-7} (K-ILS vs. schimidt), 4.32×10^{-8} (K-ILS vs. scipjack), 2.58×10^{-8} (K-ILS vs. scipjackspx), 8.48×10^{-1} (K-ILS vs. staynerd), and 4.91×10^{-1} (K-ILS against KTS), indicating that K-ILS performs similarly with the three leading algorithms (morzatball, staynerd and our previous KTS algorithm), while obviously outperforming the remaining nine algorithms.

To conclude, we observe from the tables that, on the PCSPG our K-ILS algorithm (as well as our previous KTS algorithm) is highly competitive on the 'general' instances and 'particular' instances, though it performs less well on the 'large' instances. Overall, K-ILS competes favorably with the leading competing algorithms "mozartballs" and "staynerd" [13], which are also hybrid algorithms that combine different integer programming formulations (including (x, y)-model and y-model) and different search strategies (including exact methods and heuristic search techniques) in order to effectively tackle different types of instances with different structures.

Finally, we notice that very recently some new PCSPG results are released in [19] and [32] (further improved versions of [18] and [13], respectively). However, among the 32 DIMACS challenge instances, on all the instances where our K-ILS algorithm performs the best or tied the best (in terms of *Quality Challenge*) with respect to the DIMACS competing algorithms, K-ILS still performs the best or tied the best with respect to these two improved algorithms, indicating that K-ILS is able to obtain high-quality results compared to the newest PCSPG algorithms.

4 Analysis of search components

In this section, we analyze the impact of several important search components on the performance of the proposed K-ILS algorithm, including the swap-vertex neighborhood associated with the auxiliary evaluation function, as well as the knowledge-guided perturbation strategies developed for different types of instances. All the experimental results reported in this section are executed on a personal computer with an Intel(R) Core(TM) i5-4460 3.20GHz processor (4 cores, each job occupies one core) and 4GB RAM.

4.1 Impact of the swap-vertex neighborhood and auxiliary evaluation function

As described in Section 2.8, for the 'particular' instances with uniform or nearly uniform edge costs, we combine the four basic neighborhoods in Section 2.6 with the swap-vertex neighborhood in Section 2.8.1 for local optimization, guided by an auxiliary evaluation function. In order to analyze the influences of these search mechanisms, based on the standard K-ILS algorithm described in Section 2, we implement a variant (named K-ILS-V1) by disabling the swap-vertex neighborhood and the auxiliary evaluation function while keeping the remaining ingredients unchanged. We compare the performances of K-ILS-V1 and K-ILS based on 40 'particular' PCSPG instances collected by the 11th DIMACS Challenge. Similarly, for each instance, we repeatedly independently run K-ILS-V1 (respectively, the standard K-ILS algorithm) a number of times, each independent run restarts from a randomly generated initial solution and uses the termination criterion described in Section 2.1, and the whole search process terminates until two CPU hours is elapsed. The obtained results are summarized in Table 6, where the first column indicates the instance name, the following four columns respectively indicate the overall best objective value found by K-ILS-V1, the average objective value of each independent run, the standard deviation of the objective values found by independent runs (column 'SD'), as well as the number of independent restarts. For comparisons, the next four columns show the same information corresponding to the standard K-ILS algorithm, where the objective values better than those of K-ILS-V1 are indicated in **bold**, and the same objective values are indicated in *italic*. Finally, we give the improvement percentages obtained by the standard K-ILS algorithm over those obtained by K-ILS-V1, in terms of best objective values.

As shown in Table 6, on the one hand, in terms of best objective values, the standard K-ILS algorithm obtains within the same allowed time (two CPU hours for each instance, generally corresponding to much fewer independent runs than K-ILS-V1) 13 better results compared to K-ILS-V1, and yields the same results on the remaining instances (most of which already reaching opti-

Table 6
Impact of the swap-vertex move operator and the auxiliary evaluation function (tested on 40 representative PCSPG instances with uniform or nearly uniform edge costs, each within two CPU hours).

Instance	K-ILS-V1 (Without Swap Move)				Standard K-ILS				Improve
	f^{best}	f^{avg}	SD	Restarts	f^{best}	f^{avg}	SD	Restarts	
hc6u	36.00	36.14	0.30	38337	*36.00*	**36.00**	0.00	41877	0.00%
hc7u	72.00	72.87	0.35	13671	*72.00*	**72.00**	0.00	8534	0.00%
hc8u	143.00	144.63	0.50	3513	*143.00*	**143.04**	0.14	1586	0.00%
hc9u	285.00	286.25	0.66	630	**283.00**	**283.41**	0.47	317	0.70%
hc10u	560.00	562.71	0.97	111	**559.00**	**559.47**	0.50	49	0.18%
hc11u	1118.00	1120.50	0.83	16	**1116.00**	**1116.43**	0.47	7	0.18%
hc12u	2227.00	2227.00	0.00	3	**2222.00**	**2222.50**	0.50	2	0.22%
hc6u2	20.00	20.00	0.00	62363	*20.00*	*20.00*	0.00	58121	0.00%
hc7u2	47.00	47.00	0.00	19808	*47.00*	**47.00**	0.00	8706	0.00%
hc8u2	97.00	98.24	0.38	4957	*97.00*	**97.46**	0.49	2169	0.00%
hc9u2	190.00	191.56	0.55	1039	*190.00*	**190.00**	0.00	555	0.00%
hc10u2	381.00	383.20	0.68	180	**379.00**	**380.00**	0.17	79	0.52%
hc11u2	753.00	755.32	1.23	25	**750.00**	**751.09**	1.09	11	0.40%
hc12u2	1498.00	1499.00	0.50	4	**1493.00**	**1493.00**	0.00	2	0.33%
bip42nu	227.00	228.34	0.59	197	**226.00**	**226.66**	0.49	111	0.44%
bip52nu	224.00	225.42	0.76	74	**222.00**	**222.90**	0.24	40	0.89%
bip62nu	214.00	216.43	0.88	166	*214.00*	**214.02**	0.02	83	0.00%
bipa2nu	325.00	328.35	0.70	20	*325.00*	**325.10**	0.40	10	0.00%
bipe2nu	53.00	53.49	0.50	1860	*53.00*	**53.00**	0.00	938	0.00%
cc3-4nu	10.00	10.60	0.50	56587	*10.00*	**10.58**	0.49	43864	0.00%
cc3-5nu	17.00	17.00	0.00	25253	*17.00*	*17.00*	0.00	18991	0.00%
cc3-10nu	61.00	61.01	0.14	474	*61.00*	**61.00**	0.00	275	0.00%
cc3-11nu	79.00	80.91	1.86	211	*79.00*	80.91	2.95	127	0.00%
cc3-12nu	95.00	98.18	5.27	146	*95.00*	**96.73**	3.03	81	0.00%
cc5-3nu	36.00	36.66	0.44	10164	*36.00*	**36.50**	0.50	7093	0.00%
cc6-2nu	15.00	15.00	0.00	72233	*15.00*	*15.00*	0.00	68881	0.00%
cc6-3nu	95.00	96.71	0.68	881	*95.00*	**95.19**	0.38	399	0.00%
cc7-3nu	273.00	275.09	0.78	55	**271.00**	**272.19**	0.72	26	0.73%
cc9-2nu	83.00	84.73	0.40	2251	*83.00*	**83.92**	0.47	1101	0.00%
cc10-2nu	168.00	169.52	0.75	365	*168.00*	**168.21**	0.40	160	0.00%
cc11-2nu	306.00	308.97	1.12	69	**304.00**	**306.12**	0.88	33	0.65%
cc12-2nu	569.00	570.58	1.02	12	**566.00**	**567.33**	0.47	6	0.53%
HCMV	7371.54	7372.18	0.76	46	*7371.54*	7372.28	0.82	45	0.00%
lymphoma	3341.89	3342.16	0.39	155	*3341.89*	**3342.10**	0.16	144	0.00%
metabol_expr_mice_1	11349.16	11352.88	1.59	66	*11349.16*	11353.81	3.69	32	0.00%
metabol_expr_mice_2	16250.24	16258.11	29.31	105	*16250.24*	**16256.70**	8.80	59	0.00%
metabol_expr_mice_3	16919.62	16937.47	97.78	84	*16919.62*	**16921.97**	6.93	83	0.00%
drosophila001	8275.48	8277.78	0.93	8	8277.81	8278.09	0.50	6	-0.03%
drosophila005	8128.32	8129.90	0.24	6	**8126.99**	**8128.20**	0.94	5	0.02%
drosophila0075	8047.77	8049.55	3.20	5	8048.11	**8049.37**	0.34	5	-0.00%

mality [13]) only with 2 exceptions. This corresponds to a mean improvement percentage of 0.14% (averaged on all these 40 instances) by the standard K-ILS algorithm over K-ILS-V1. Furthermore, we use the Wilcoxon test to check the statistical difference between the best objective values of the standard K-ILS algorithm and those of K-ILS-V1, which reveals a p-value of 4.51×10^{-3}, confirming that the difference is statistically significant.

On the other hand, in terms of average objective values, the standard K-ILS algorithm respectively obtains 33 better, 4 same and 3 worse results with respect to K-ILS-V1. Once again, the Wilcoxon test reveals a p-value of 9.49×10^{-7}, indicating an even more significant difference.

Note that the only difference between the standard K-ILS and K-ILS-V1 is the use of the swap-vertex neighborhood associated with the auxiliary evaluation function, this comparison clearly demonstrates the importance of these search components to the effectiveness of the proposed K-ILS algorithm.

4.2 Impact of the knowledge-guided perturbation operators

As mentioned in Section 2, K-ILS relies on several knowledge-guided perturbation strategies to escape from local optima, respectively developed for tackling the 'general' and 'particular' instances. To analyze the impacts of these perturbation strategies, in this subsection we implement another adapted variant (named K-ILS-V2) of the standard K-ILS algorithm, which uses the randomized initialization method described in Section 2.5 to generate a new solution whenever a perturbation is needed, while keeping all the remaining ingredients and parameters in accordance with the standard K-ILS algorithm.

Table 7
Impact of the knowledge-guided perturbation strategy (tested on the 29 RPCST instances used in the 11th DIMACS Challenge, each within two CPU hours).

Instance	K-ILS-V2 (Random Restart)				Standard K-ILS				Improve
	f^{best}	f^{avg}	SD	Restarts	f^{best}	f^{avg}	SD	Restarts	
i101M1	109271.50	109271.50	0.00	18348	*109271.50*	*109271.50*	0.00	18976	0.00%
i101M2	315925.31	316665.19	317.91	5084	*315925.31*	**316489.91**	278.13	4309	0.00%
i101M3	355942.41	358460.56	1818.57	1928	**355625.41**	**356404.53**	1539.34	2124	0.09%
i102M1	104065.80	104065.80	0.00	18180	*104065.80*	*104065.80*	0.00	18754	0.00%
i102M2	352538.81	354700.81	709.95	11023	*352538.81*	**352876.31**	675.31	3638	0.00%
i102M3	455055.94	455657.81	75.46	10166	**454365.94**	**454999.28**	266.65	2804	0.15%
i103M1	139749.41	139749.41	0.00	18149	*139749.41*	*139749.41*	0.00	18698	0.00%
i103M2	407834.22	408318.75	204.03	7406	*407834.22*	**408164.22**	236.74	5763	0.00%
i103M3	456678.50	457380.88	99.89	3402	**456125.50**	**457115.69**	388.77	3148	0.12%
i104M2	89920.84	89920.84	0.00	17645	*89920.84*	*89920.84*	0.00	18363	0.00%
i104M3	97148.79	97148.79	0.00	14784	*97148.79*	*97148.79*	0.00	15014	0.00%
i105M1	26717.20	26717.20	0.00	18233	*26717.20*	*26717.20*	0.00	18596	0.00%
i105M2	100269.62	100269.62	0.00	15112	*100269.62*	*100269.62*	0.00	15552	0.00%
i105M3	110351.16	112629.95	564.27	13267	*110351.16*	**112024.70**	1139.68	8601	0.00%
i201M2	355467.69	355467.69	0.00	3358	*355467.69*	*355467.69*	0.00	3608	0.00%
i201M3	629353.38	632896.75	516.81	2057	**628833.63**	**630780.94**	1623.50	1178	0.08%
i201M4	773848.31	777045.56	1418.24	2487	**773398.31**	**775561.44**	1315.04	1223	0.06%
i202M2	288946.84	288946.84	0.00	3108	*288946.84*	*288946.84*	0.00	3541	0.00%
i202M3	419287.88	419344.94	7.96	2344	**419184.16**	**419206.81**	33.08	1137	0.02%
i202M4	430113.31	455355.50	9507.86	2430	**430034.25**	**432274.38**	22.25	822	0.02%
i203M2	459894.78	459894.78	0.00	3367	*459894.78*	*459894.78*	0.00	3537	0.00%
i203M3	643062.00	643255.19	89.89	2147	*643062.00*	**643225.88**	92.68	1149	0.00%
i203M4	677733.06	679023.25	743.87	702	*677733.06*	**677929.38**	170.13	648	0.00%
i204M2	161700.55	161700.55	0.00	5470	*161700.55*	*161700.55*	0.00	5709	0.00%
i204M3	245287.20	245387.42	24.45	2299	*245287.20*	**245330.84**	10.01	1248	0.00%
i204M4	245287.20	245355.16	20.04	2150	*245287.20*	245358.84	63.88	2051	0.00%
i205M2	571186.00	571367.94	15.10	1030	**571031.44**	571386.81	136.12	1966	0.03%
i205M3	672646.69	672926.50	25.30	1084	**672403.13**	**672521.44**	62.89	855	0.04%
i205M4	714292.25	714498.63	71.86	972	**713973.63**	**714020.75**	76.56	758	0.04%

First, we compare the performances of the K-ILS-V2 variant and the standard K-ILS algorithm on the 29 RPCST instances used in Section 3.2. Like in the last subsection, for each instance, we repeatedly run K-ILS-V2 (respectively, the standard K-ILS algorithm) until the cutoff of two CPU hours is elapsed. The results of this experiment are shown in Table 7, with the same information as in Table 6. Table 7 discloses that, in terms of best objective values, the standard K-ILS algorithm obtains 10 better results compared to K-ILS-V2, and yields the same results on all the remaining 19 instances (all reaching optimality), corresponding to an average improvement percentage of 0.02% over K-ILS-V2. The Wilcoxon test reveals a p-value of 1.57×10^{-3}, indicating a statistically significant difference. In terms of average objective values, the standard K-ILS respectively obtains 16 better, 11 same and 2 worse results

compared to K-ILS-V2, corresponding to a p-value of 9.67×10^{-4}, indicating again a significant difference.

Table 8
Impact of the knowledge-guided perturbation strategies on 26 'general' or 'particular' PCSPG instances of the 11th DIMACS Challenge (each within two CPU hours).

Instance	K-ILS-V2 (Random Restart)				Standard K-ILS				Improve
	f^{best}	f^{avg}	SD	Restarts	f^{best}	f^{avg}	SD	Restarts	
C13-A	236.00	237.49	0.66	2527	*236.00*	**236.75**	0.50	2502	0.00%
C19-B	146.00	146.74	0.54	434	*146.00*	**146.02**	0.28	493	0.00%
D03-B	1512.00	1520.98	2.82	1787	**1509.00**	**1511.86**	2.58	1355	0.20%
D20-A	536.00	536.00	0.00	87	*536.00*	*536.00*	0.00	91	0.00%
P400_3	2951725.00	2955812.25	1509.76	3146	*2951725.00*	2957188.75	3279.15	2705	0.00%
P400_4	2852956.00	2856611.50	2898.11	3376	*2852956.00*	2857842.75	4337.09	2545	0.00%
K400_7	474466.00	480920.06	277.20	11084	*474466.00*	**480107.09**	2220.79	8783	0.00%
K400_10	395310.00	401478.41	1507.44	8998	**394191.00**	**399901.97**	3251.63	7319	0.28%
i640-001	2932.00	2935.84	3.43	20941	*2932.00*	2951.49	112.74	19411	0.00%
i640-221	8400.00	8400.16	1.08	158	*8400.00*	**8400.00**	0.00	155	0.00%
i640-321	28787.00	28787.15	0.28	47	*28787.00*	**28787.02**	0.02	60	0.00%
i640-341	29689.00	29741.04	26.27	155	**29671.00**	29742.13	32.29	135	0.06%
a2000RandGraph_2	1490.49	1491.15	0.03	10	**1489.75**	**1490.79**	0.63	9	0.05%
a4000RandGraph_3	3412.51	3413.51	1.00	2	3413.41	3413.57	0.16	2	-0.03%
hc10p	60133.00	60307.04	62.96	27	**59778.00**	**59948.81**	77.47	47	0.59%
hc11u	1131.00	1132.33	3.00	3	**1116.00**	**1116.71**	0.50	7	1.33%
hc12p	238815.00	238815.00	0.00	1	**236435.00**	**236435.00**	0.00	1	1.00%
hc12u	2266.00	2266.00	0.00	1	**2222.00**	**2223.00**	1.00	2	1.94%
bip52nu	225.00	227.87	1.01	39	**222.00**	**222.92**	0.28	38	1.33%
bip62nu	214.00	215.69	0.73	58	*214.00*	**214.00**	0.00	83	0.00%
cc3-12nu	95.00	95.71	0.41	96	*95.00*	96.63	3.16	86	0.00%
cc12-2nu	574.00	575.50	1.50	2	**566.00**	**567.33**	0.50	6	1.39%
metabol_expr_mice_1	11349.70	11352.18	0.75	44	**11349.16**	11353.77	4.34	33	0.00%
HCMV	7371.54	7371.54	0.00	158	*7371.54*	7372.34	0.90	39	0.00%
lymphoma	3341.89	3341.96	0.16	202	*3341.89*	3342.11	0.41	133	0.00%
drosophila001	8276.50	8278.24	0.66	50	8277.81	**8278.09**	0.27	6	-0.02%

Furthermore, we compare the performances of K-ILS-V2 and the standard K-ILS algorithm on the 26 'general' or 'particular' PCSPG instances adopted by the 11th DIMACS challenge. As shown in Table 8, in terms of best objective values, the standard K-ILS algorithm obtains 11 better and 2 worse results compared to K-ILS-V2, and yields the same results on the remaining 13 instances. The average improvement percentage of the standard K-ILS algorithm over K-ILS-V2 is 0.31%, and the Wilcoxon test reveals a p-value of 1.26×10^{-2}, indicating a statistically better performance of the standard K-ILS algorithm over K-ILS-V2. In terms of average objective values, the standard K-ILS algorithm respectively obtains 16 better, 1 same and 9 worse results, corresponding to a p-value of 1.62×10^{-1}, indicating the statistical difference is not so significant.

To conclude, above comparisons confirm clearly the interest of the knowledge-guided perturbation strategies for solving both the RPCST and the PCSPG.

5 Conclusions

The prize-collecting Steiner tree problem in graphs (PCSPG) and its rooted version (RPCST) are both target problems of the recent 11th DIMACS Im-

plementation Challenge (2014) [29]. In this paper, we proposed a knowledge-guided iterated local search approach named K-ILS for solving both the PC-SPG and the RPCST. K-ILS combines several key search strategies, including a swap-vertex move operator associated with an auxiliary evaluation function which are shown to be very useful for some extremely challenging instances with uniform or nearly uniform edge costs, as well as several knowledge-guided perturbation strategies, which are highly effective on various types of instances. We also designed two original path-based move operators. Experimental results showed that K-ILS is able to improve our previous KTS algorithm [15], which was ranked the first place or tied the first place on five of the eight main subcategories where it was involved during the 11th DIMACS Implementation Challenge. Finally, we performed additional experiments to investigate the impact of several key components over the performance of the proposed algorithm.

One notices that the idea of the swap-vertex operator could be adapted to several other Steiner tree problems, such as the classical Steiner tree problem and the maximum-weight connected subgraph problem. Finally, this work confirms that it is a good practice to consider characteristics of problem instances when designing search strategies. Such a design could be applied to other settings where relevant instance features can be identified and effectively explored by search procedures.

Acknowledgments

We are grateful to the anonymous referees for their valuable suggestions and comments which helped us to improve the paper. This work was partially supported by the following projects: LigeRO (Pays de la Loire Region, France), PGMO (Jacques Hadamard Mathematical Foundation, Paris, France), a post-doc fellowship from Angers Loire Metropole, and the National Natural Science Foundation of China (grant no: U1613216). We would like to thank the organizers of the 11th DIMACS Implementation Challenge for organizing the competition and the workshop, and give our special thanks to Dr. Gerald Gamrath for kindly helping us to evaluate our algorithm on the computing platform of the Challenge.

References

[1] Akhmedov, M., Kwee, I., Montemanni, R. 2016. A divide and conquer matheuristic algorithm for the prize-collecting Steiner tree problem. *Computers and Operations Research*, 70, 18-25.

[2] Althaus, E., Blumenstock, M. 2014. Algorithms for the maximum weight connected subgraph and prize-collecting Steiner tree problems. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4-5, 2014, `http://dimacs11.zib.de/workshop.html`

[3] Archer, A., Bateni, M., Hajiaghayi, M., Howard Karloff, H. 2011. Improved approximation algorithms for prize-collecting Steiner tree and TSP. *SIAM Journal on Computing*, 40(2), 309-332.

[4] Berthold, T. 2013. Measuring the impact of primal heuristics. *Operations Research Letters*, 41, 611-614.

[5] Biazzo, I., Braunstein, A., Zecchina, R. 2012. On the performance of a cavity method based algorithm for the prize-collecting Steiner tree problem on graphs. *Physical Review: E*, 86, 026706.

[6] Bienstock, D., Goemans, M. X., Simchi-Levi, D., Williamson, D.P. 1993. A note on the prize collecting travelling salesman problem. *Mathematical Programming*, 59, 413-420.

[7] Canuto, S. A., Resende, M. G. C., Ribeiro, C. C. 2001. Local search with perturbations for the prize-collecting Steiner tree problem in graphs. *Networks*, 38, 50-58.

[8] Chen, K. H. 2012 Dynamic randomization and domain knowledge in Monte-Carlo tree search for Go knowledge-based systems. *Knowledge-Based Systems*, 34(5), 21-25.

[9] Constantinou, C. K., Ellinas, G., Panayiotou, C. 2015. A low-complexity binary-heap implementation of Dijkstra's algorithm. *Computer Science*.

[10] El-Kebir, M., Klau, G. W. 2014. Solving the maximum-weight connected subgraph problem to optimality. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4-5, 2014, available online at `http://dimacs11.zib.de/workshop.html`

[11] Engevall, S., Göthe-Lundgren, M., Värbrand, P. 1998. A strong lower bound for the node weighted Steiner tree problem. *Networks*, 31, 11-17.

[12] Feofiloff, P., Fernandes, C. G., Ferreira, C. E., Pina, J. C. D. 2007. Primal-dual approximation algorithms for the prize-collecting Steiner tree problem. *Information Processing Letters*, 103, 195-202.

[13] Fischetti, M., Leitner, M., Ljubic, I., Luipersbeck, M., Monaci, M., Resch, M., Salvagnin, D., Sinnl, M. 2014. Thinning out Steiner trees: a node-based model for uniform edge costs. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4-5, 2014, available online at `http://dimacs11.zib.de/workshop.html`

[14] Fu, Z. H., Hao, J. K. 2014. Breakout local search for the Steiner tree problem with revenue, budget and hop constraints. *European Journal of Operational Research*, 232(1), 209-220.

[15] Fu, Z. H., Hao, J. K. 2014. Knowledge guided tabu search for the prize-collecting Steiner tree problem in graphs. Presented at the workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4-5, 2014, available online at `http://dimacs11.zib.de/workshop.html`

[16] Fu, Z. H., Hao, J. K. 2015. Dynamic programming driven memetic search for the Steiner tree problem with revenue, budget and hop constraints. *INFORMS Journal on Computing*, 27(2), 221-237.

[17] Fu, Z. H., Hao, J. K. 2016. Swap-vertex based neighborhood for Steiner tree problems. *Mathematical Programming Computation* (in press). DOI 10.1007/s12532-016-0116-8

[18] Gamrath, G., Koch, T., Maher, S. T., Rehfeldt, D., Shinano, Y. 2014. SCIP-Jack - A solver for STP and variants with parallelization extensions. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4-5, 2014, available at `http://dimacs11.zib.de/workshop.html`

[19] Gamrath, G., Koch, T., Maher, S. T., Rehfeldt, D., Shinano, Y. 2016. SCIP-Jack - A solver for STP and variants with parallelization extensions. *Accepted to Mathematical Programming Computation*, available online at `https://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/6017`.

[20] Goemans, M. X., Williamson, D. P. 1995. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2), 296-317.

[21] Goemans, M. X., Williamson, D. P. 1997. The primal-dual method for approximation algorithms and its application to network design problems. In D. S. Hochbaum editor, Approximation Algorithms for NP-Hard Problems, PWS Publiching Company, Boston, 144-191.

[22] Glover, F., Laguna, M. 1997. Tabu search. Kluwer Academic Publishers.

[23] Goldbarg, E. F. G., Goldbarg, M. C., Schmidt, C. C. 2008. A hybrid transgenetic algorithm for the prize collecting Steiner tree problem. *Journal of Universal Computer Science*, 14(15), 2491-2511.

[24] Grunskiï, I. S., Maksimenko, I. I. 2012. Interpretable knowledge extraction from emergency call data based on fuzzy unsupervised decision tree. *Knowledge-Based Systems*, 25(1), 77-87.

[25] Hakimi, S. L. 1971. Steiner's problem in graphs. *Networks*, 1, 113-133.

[26] Hegde, C., Indyk, P., Schmidt, L. 2014. A Fast, adaptive variant of the Goemans-Williamson scheme for the prize-collecting Steiner tree problem. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4-5, 2014, available online at `http://dimacs11.zib.de/workshop.html`

[27] Hoos, H. H., Stützle T. 2004. Stochastic local search. *Foundations and Applications*, Morgan Kaufmann.

[28] Johnson, D. S., Minkoff, M., Phillips, S. 2000. The prize collecting steiner tree problem: theory and practice. Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 760-769.

[29] Johnson, D. S., Koch, T., Werneck, R. F., Zachariasen, M. 2013-2014. 11th DIMACS Implementation Challenge in collaboration with ICERM: Steiner tree problems. `http://dimacs11.zib.de`

[30] Karp, R. M. 1972. Reducibility among combinatorial problems. Miller, R. E., Thatcher, J. W., eds. *Complexity of Computer Computations*, New York.

[31] Klau, W., Ljubić I., Moser, A., Mutzel, P., Neuner, P., Pferschy, U., Raidl, G., Weiskircher, R. 2004. Combining a memetic algorithm with integer programming to solve the prize collecting Steiner tree problem. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004), Lecture Notes in Computer Science*, 3102, 1304-1315.

[32] Leitner, M., Ljubić I., Luipersbeck, M., Sinnl, M. 2016. A dual-ascent-based branch-and-bound framework for the prize-collecting Steiner tree and related problems. Technical report, available online at `http://www.optimization-online.org/DB_HTML/2016/06/5509.html`.

[33] Ljubić, I., Weiskircher, R., Pferschy, U., Klau, G., Mutzel, P., Fischetti, M. 2006. An algorithmic framework for the exact solution of the prize-collecting Steiner tree problem. *Mathematical Programming: Series B*, 105, 427-449.

[34] Lucena, A., Resende M. G. C. 2004. Strong lower bounds for the prize collecting Steiner problem in graphs. *Discrete Applied Mathematics*, 141(1-3), 277-294.

[35] Minkoff, M. 2000. The prize collecting Steiner tree problem. *Master's thesis*, Department of Electrical Engineering and Computer Science, Massachusetts Insitute of Technology, USA.

[36] Salles da Cunha, A., Lucena, A., Maculan, N., Resende, M. G. C. 2009. A relax-and-cut algorithm for the prize-collecting Steiner problem in graphs. *Discrete Applied Mathematics*, 157, 1198-1217.

[37] Segev, A. 1987. The node weighted Steiner tree problem. *Networks*, 17, 1-17.

[38] Uchoa, E. 2006. Reduction tests for the prize-collecting Steiner problem. *Operations Research Letters*, 34, 437-444.

[39] Uchoa, E., Werneck, R. F. F. 2010. Fast local search for Steiner trees in graphs. *In Blelloch, G.E., Halperin, D., eds.: ALENEX, SIAM*, 1-10.

[40] Wolpert, D. H., Macready, W. G. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), 67-82.

A Appendix: Parameters tuning

We discuss how we tuned the parameters of the K-ILS algorithm. Given that the benchmark instances have very different structures, it is extremely difficult to obtain a set of parameter values which yield uniformly the best result on all instances. Thus we try to determine a parameter setting which leads to a globally good performance. As indicated in Section 3.1, our experiments show that the default setting of Table 1 with L_{flip}=n, L_{swap}=0.5n, M=30 and W=50 performs reasonably well on each group of instances (parameter L_{flip} is applicable only for the general instances and parameter L_{swap} is applicable only for the particular instances). In order to determine these values, we implemented eight scenarios for each parameter by varying the chosen parameter within a reasonable range (shown in Table A.1), while fixing the other parameters in accordance with the default values of Table 1.

We used the 32 PCSPG challenging instances (Table 4, including 14 'general' instances, 12 'particular' instances and 6 'large' instances) as sample instances to evaluate the performances of the compared scenarios. For each parameter, we only used the instances on which the performance of K-ILS might be affected by this parameter, i.e. the 14 'general' instances to tune L_{flip}, and the 12 'particular' instances to tune L_{swap}, and all 32 instances to tune M and W.

To solve each test instance, we repeatedly and independently ran the algorithm with each scenario until a cutoff time of one CPU hour was reached, on a laptop computer with an Intel(R) Core(TM) i5-4460 3.20GHz processor and 4GB RAM. We recorded the best-found objective value of each scenario, and then calculated the points collected by each scenario in terms of *Quality Challenge* (defined as *Formula 1* at the beginning of Section 3), with respect to the remaining seven scenarios corresponding to the studied parameter. As explained in Section 3, a scenario collecting more points indicates a better overall performance (based on the 32 test instances) with respect to other compared scenarios. The experimental results are given in Table A.1, being classified into four groups, each corresponding to a particular parameter. For each parameter, the column "Value" indicates the different settings of the studied parameter, while the column "Points" gives the points collected by the corresponding scenario. In addition, we used the Friedman test to check the statistical differences between the competing scenarios.

Parameter L_{flip}: We varied L_{flip} within the range $[0.1n, 5n]$ to get eight scenarios and recorded the results in Table A.1 (columns 1-2), which shows that the scenario with L_{flip}=n collects the most points (56.75) with respect to the other seven scenarios. The Friedman test reveals a p-value of 2.58×10^{-1}, indicating that K-ILS is not really sensitive to the parameter L_{flip}.

Table A.1
Statistical results based on 32 challenging PCSPG instances corresponding to different parameter values

Parameter L_{flip}		Parameter L_{swap}		Parameter M		Parameter W	
Value	Points	Value	Points	Value	Points	Value	Points
$0.1n$	47.75	$0.1n$	31.38	10	105.88	10	89.46
$0.2n$	48.75	$0.2n$	46.05	20	90.63	20	108.16
$0.3n$	49.75	$0.3n$	45.63	30	**141.85**	30	113.41
$0.5n$	43.25	$0.5n$	**50.13**	50	110.21	50	**127.36**
n	**56.75**	n	37.80	100	93.83	100	94.06
$2n$	38.08	$2n$	26.71	200	99.55	200	100.15
$3n$	42.08	$3n$	43.80	300	106.00	300	97.45
$5n$	37.58	$5n$	30.50	500	84.07	500	101.93

Parameter L_{swap}: Similarly, we tested L_{swap} within the range $[0.1n, 5n]$ and tested their performances. The results in Table A.1 (columns 3-4) indicate that the scenario with L_{swap}=$0.5n$ performs overall the best (collects 50.13 points) among the eight different scenarios. The Friedman test reveals a p-value of 2.71×10^{-1}, indicating that L_{swap} is not a sensitive parameter.

Parameter M: Furthermore, we varied M within the range $[10, 500]$ with variable steps and compared their performances. As shown in Table A.1 (columns 5-6), the scenario with M=30 collects the most points (141.85). The Friedman test reveals a p-value of 9.02×10^{-4}, confirming that the scenario with M=30 performs statistically clearly better than other scenarios.

Parameter W: Finally, we tested M within the range $[10, 500]$ to get eight scenarios and listed their results in Table A.1 (columns 7-8). The results indicate that the scenario with W=50 performs slightly better than other scenarios. The Friedman test reveals a p-value of 8.08×10^{-2}, indicating that the performance of K-ILS is somewhat sensitive to the value of this parameter and W=50 is the best setting.