CrossMark

# Swap-vertex based neighborhood for Steiner tree problems

**Zhang-Hua Fu[1,2]** · **Jin-Kao Hao[2,3]**

**Abstract** Steiner tree problems (STPs) are very important in both theory and practice. In this paper, we introduce a powerful swap-vertex move operator which can be used as a basic element of any neighborhood search heuristic to solve many STP variants. Given the incumbent solution tree $T$, the swap-vertex move operator exchanges a vertex in $T$ with another vertex out of $T$, and then attempts to construct a minimum spanning tree, leading to a neighboring solution (if feasible). We develop a series of dynamic data structures, which allow us to efficiently evaluate the feasibility of swap-vertex moves. Additionally, in order to discriminate different swap-vertex moves corresponding to the same objective value, we also develop an auxiliary evaluation function. We present a computational assessment based on a number of challenging problem instances (corresponding to three representative STP variants) which clearly shows the effectiveness of the techniques introduced in this paper. Particularly, as a key element of our KTS algorithm which participated in the 11th DIMACS implementation challenge, the swap-vertex operator as well as the auxiliary evaluation function contributed significantly to the excellent performance of our algorithm.

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

   Zhang-Hua Fu
   fu@info.univ-angers.fr

[1] Institute of Robotics and Intelligent Manufacturing, The Chinese University of Hong Kong, Shenzhen, Shenzhen 518172, China

[2] LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers Cedex 01, France

[3] Institut Universitaire de France, Paris, France

🖉 Springer

## 1 Introduction

Many network design problems encountered in real-life applications (e.g., electricity,
telecommunication, transportation, distribution supply, VLSI design, biology, signal
processing, etc.) require to connect a number of basic equipments, with respect to
various objectives and constraints [1]. These problems could generally be formulated
as a class of broadly defined Steiner tree problems (STPs), where the optimal solutions
have the structure of a tree. According to the pursued objectives and/or constraints to
be considered, a number of STP variants can be defined, e.g., the classical STP [2],
the rectilinear STP [3], the Euclidean STP [4], the prize-collecting STP [5], the node
weighted STP [6] (including the maximum weight connected subgraph problem [7]),
the group STP [8], the generalized STP [9], the obstacle avoiding STP [10], the directed
STP [11], the dynamic STP [12], the stochastic STP [13], the STP with hop constraints
[14], and mix of them [15,16], etc. Notice that almost all these STP variants are known
to be NP-hard [17–19] and thus computationally challenging.

Given the theoretical importance and practical relevance of the STPs, considerable
effort has been put forth to develop both exact and heuristic methods during the last
decades. On the one hand, numerous powerful exact algorithms (based on branch
and bound, branch and cut, branch and price, etc.) have been devised. These methods
have the desirable property of guaranteeing to find the optimal solution if no time
and space limit is imposed. In practice, the current best exact algorithms are able
to achieve remarkable results on several well-studied STP variants [20–23]. On the
other hand, due to the inherent computational complexity of STPs, in some cases, it is
impossible for a method to enumerate effectively the solutions of a given instance. To
handle problems whose optimal solutions cannot be reached within a reasonable time,
heuristic and meta-heuristic algorithms, which aim to provide sub-optimal solutions
within an acceptable time, have become the mainly applied methods. Actually, exact
and heuristic methods complement each other and could be used to tackle different
types of problems. These two approaches can also be combined to create even more
powerful hybrid methods.

Among various heuristics for solving the STPs, local search or neighborhood search
is certainly the most popular and effective approach [24–29]. Generally, local search
relies on some key ingredients including most importantly one or several move (or
transformation) operators responsible for generating neighboring solutions [30]. In the
field of STPs, a couple of conventional move operators have already been developed.
For instance, for the classical STP in graphs (SPG), four basic move operators, i.e.,
vertex insertion, vertex elimination, key-path exchange, key-vertex elimination are
commonly used in the literature [24,25], while for the prize-collecting STP (PCSPG),
two move operators called vertex insertion and vertex elimination were developed in

[27]. These move operators add or remove vertices and then try to reconstruct a new minimum spanning tree, since for both the SPG and the PCSPG, each feasible solution could be uniquely characterized by its spanned vertices set. Though the existing move operators are generally quite effective, additional improvements are still possible by introducing new move operators.

This paper is interested in designing a new and effective move operator, i.e., the swap-vertex move operator, which complements the existing move operators, and can be adopted by local search heuristics to solve a large class of STP variants. Given the incumbent solution tree $T$, the swap-vertex move operator exchanges a vertex in $T$ with another vertex out of $T$, and then tries to construct a minimum spanning tree (MST), resulting in a neighboring solution of $T$. This basic move operator realizes a natural operation and could be advantageously employed by heuristic methods. However, due to its unaffordable complexity in the general case, the power of this operator was not really explored by the existing heuristics designed for solving STPs.

In this work, we demonstrate that the swap-vertex move operator could be effectively applied to instances of various STP problems when (1) the optimal solution of the problem is necessarily a minimum spanning tree once the vertices to span are known, (2) the input graph has uniform edge costs (i.e., all the edges have the same cost) and (3) the input graph is of reasonable size.[1] This covers all the MWCS (the maximum weight connected subgraph problem) instances, as well as a number of particularly difficult SPG and PCSPG instances collected by the 11th DIMACS challenge (detailed in Sect. 4). More generally, as explained in [28], the proposed swap-vertex move operator can also be slightly adapted to solve problem instances with nearly uniform edge costs, even if its effectiveness might be somewhat impacted.

Another contribution of this work is a newly designed auxiliary evaluation function (defined in Sect. 3.3), based on the concept of special degree of a given solution. The auxiliary evaluation function aims to identify the most promising swap-vertex move among a number of candidate moves with the same move gain in terms of the objective function. The new function is used to guide the search algorithm towards a promising direction on search plateaus when the objective function alone cannot distinguish the candidate moves.

The remainder of this paper is organized as follows: After introducing three representative STP variants in Sect. 2, Sect. 3 presents the main idea as well as the technical details relative to the swap-vertex move operator. Section 4 reports experimental results on these three STP variants, in order to demonstrate the generality and effectiveness of the swap-vertex move operator. Finally, Sect. 5 is dedicated to concluding remarks.

## 2 Problem definitions

To illustrate the main idea of the swap-vertex based neighborhood and the associated evaluation technique for Steiner tree problems, we choose, as illustrative examples,

---

[1] Depending on the available memory of the used computer, currently the swap-vertex move operator has been successfully applied to solve instances with up to 5226 vertices and 93,394 edges, executed on a personal computer with 4 GB RAM.

three representative STP variants: the classical Steiner tree problem in graphs (SPG), the prize-collecting Steiner tree problem in graphs (PCSPG), and the maximum weight connected subgraph problem (MWCS). These STP variants are part of the 11th DIMACS implementation challenge [31] and attracted a large number of participants during the challenge.

### 2.1 Classical Steiner tree problem in graphs (SPG) [2]

Given an undirected graph $G = (V, E)$ with a set $V$ of vertices and a set $E$ of edges. The set $V$ is partitioned into two sets: a set of terminal vertices and a set of Steiner vertices. Each edge $e \in E$ has an associated cost $c_e \geq 0$. The SPG is to determine a subtree $T = (V_T, E_T)$ (with vertex set $V_T$ and edge set $E_T$ respectively) of $G$ spanning all terminal vertices and possibly some Steiner vertices, so as to minimize the total edge cost of the obtained tree, i.e.:

$$Minimize\ f_1(T) = \sum_{e \in E_T} c_e. \tag{1}$$

Specifically, the rectilinear Steiner tree problem [3] can be reduced to a special case of the SPG.

### 2.2 Prize-collecting Steiner tree problem in graphs (PCSPG) [5]

Given an undirected graph $G = (V, E)$ with vertex set $V$ and edge set $E$. Each vertex $v \in V$ is associated with a real-valued prize $p_v \geq 0$ (vertex $v$ is called a customer vertex if $p_v > 0$, and a non-customer vertex otherwise), and each edge $e \in E$ is associated with a real-valued cost $c_e \geq 0$. Then the PCSPG aims to find a subtree $T = (V_T, E_T)$ of $G$, so as to minimize the sum of the consumed edge costs plus the prizes of the vertices not spanned by $T$, i.e.:

$$Minimize\ f_2(T) = \sum_{e \in E_T} c_e + \sum_{v \notin V_T} p_v. \tag{2}$$

Note that the SPG is a special case of the PCSPG, if each terminal of the SPG is associated with a high enough prize (corresponding to a customer of the PCSPG), and each Steiner vertex is associated with a zero prize (corresponding to a non-customer vertex of the PCSPG).

Particularly, if an additional source vertex is chosen as the root which must be part of any feasible solution, we obtain a rooted version of the PCSPG (denoted by RPCST, another of the seven competition problems included in the 11th DIMACS challenge).

### 2.3 Maximum-weight connected subgraph problem (MWCS) [7]

Given an undirected graph $G = (V, E)$ with vertex set $V$ and edge set $E$. Each vertex is associated with a real-valued weight $w_v$ ($w_v$ might be negative), then the MWCS

is to find a subset $V^* \subseteq V$ such that the induced graph over $V^*$ is connected and the total weight is maximized, i.e.:

$$Maximize \; f_3(T) = \sum_{v \in V^*} w_v. \tag{3}$$

In the MWCS, the edge costs are no longer under consideration, thus we should just guarantee connectivity of the induced graph. From the point of view of STPs, without changing the optimal solution, we can assume each edge has a zero cost, and the task is to find a subtree of graph $G$ such that the spanned vertices lead to the largest weight.

Using the rules described in [32], every PCSPG instance can be transformed to an equivalent MWCS instance, indicating that the MWCS is a basic model which potentially embraces all the four STP variants mentioned above.

## 3 Swap-vertex based neighborhood

In this section, we present the general idea of the swap-vertex move operator and devise dedicated techniques for fast neighborhood evaluation. We also define an auxiliary evaluation function which aims to distinguish different swap-vertex moves corresponding to the same objective value.

### 3.1 Main idea

According to the problem definitions, the above three STP variants (SPG, PCSPG and MWCS) share a common feature: the optimal solution must be a minimum spanning tree (MST) once the vertices to span are known. It indicates that any solution $T$ could be uniquely characterized by its spanned vertex set $V_T$. Accordingly, two basic move operators, which simply add (or remove) a vertex to (from) the incumbent solution (a MST) and then attempt to reconstruct a new MST, have been applied respectively to the SPG [25] and the PCSPG [27] (to our knowledge, no such basic move operator has been applied to the MWCS). Based on these two basic move operators, one can define two neighborhoods [denoted by $N_1(T)$, $N_2(T)$ respectively] for a given incumbent solution $T$ with vertex set $V_T$:

$$N_1(T) = \{\text{MST}(V_T \cup \{i\}), \quad \forall \, i \notin V_T\},$$
$$N_2(T) = \{\text{MST}(V_T \backslash \{i\}), \quad \forall \, i \in V_T\}. \tag{4}$$

where $\text{MST}(V_T \cup \{i\})$ and $\text{MST}(V_T \backslash \{i\})$ respectively denote the neighboring solution of $T$ achieved by adding a vertex $i$ to the vertex set $V_T$ of $T$ and removing $i$ from $V_T$.

The add and remove move operators are simple to implement and are widely adopted by local search heuristics. However, as we show in this work, the search performance of these two basic move operators could be considerably improved by introducing the swap-vertex move operator.

Basically, given the incumbent solution $T$ (a MST) with its vertex set $V_T$, the swap-vertex move operator exchanges a vertex $i \in V_T$ with another vertex $j \notin V_T$, and then

tries to reconstruct (dynamically) a new MST, leading to a neighboring solution of $T$ (possibly discarded if the new solution is infeasible). One notices that applying this move operator is equivalent to adding vertex $j$ and removing vertex $i$ simultaneously.

Accordingly, we can define the neighborhood $N_3(T)$ as follows:

$$N_3(T) = \{\text{MST}(V_T \cup \{j\}\backslash\{i\}), \quad \forall\, i \in V_T,\, j \notin V_T\}. \tag{5}$$

By combining this new neighborhood $N_3(T)$ with the two basic neighborhoods $N_1(T)$ and $N_2(T)$, a local search procedure could search within an enlarged neighborhood, thus increasing the opportunity of finding solutions of improved quality.

Although this idea is natural, the swap neighborhood has not been widely applied in existing STP local search methods, possibly due to the unreasonably high complexity needed for neighborhood exploration. Indeed, given the incumbent solution $T$ with its vertex set $V_T$, and let $n = |V|$ and $m = |E|$, then there are in total $O(|V_T|) \cdot O(|V| - |V_T|) \leq O(n^2)$ possible swap-vertex moves. If we choose to reconstruct a MST from scratch (e.g., using Kruskal's algorithm with the aid of a Fibonacci heap) after applying a swap-vertex move, the overall complexity would reach $O(n^2) \cdot O(m + n \cdot \log n)$, being unaffordable for mid- and large- sized instances.

Fortunately, for graphs with uniform edge costs, the above complexity could be reduced much. For example, in the MWCS problem which potentially covers many other STP variants, swapping a vertex $i \in V_T$ with a vertex $j \notin V_T$ (if feasible) would definitely increase the objective value by $\Delta = w_j - w_i$ ($\Delta$ is called the move gain). Furthermore, for any SPG or PCSPG instance with uniform edge costs, feasibly swapping $i \in V_T$ with $j \notin V_T$ would never change the total consumed cost, indicating that the objective value would definitively increase by $\Delta = 0$ (for the SPG) or $\Delta = p_i - p_j$ (for the PCSPG). In these cases, it is quite easy to calculate all the $O(n^2)$ possible $\Delta$ values, within an overall complexity of $O(n^2)$. However, to implement the swap operator effectively, we need to consider two additional questions: (1) How to verify the solution feasibility after swapping any pair of vertices? (2) How to distinguish the feasible moves leading to the same objective value? In the following subsections, we discuss in detail how to address these two questions.

### 3.2 Feasibility verification

Given a solution $T$ with vertex set $V_T$ and edge set $E_T$, before swapping any pair of vertices (for the SPG, we only consider swapping Steiner vertices), we should verify at first its feasibility. Without loss of generality, we assume the incumbent solution $T$ has at least two vertices, then the feasibility after swapping each pair of vertices could be efficiently verified, with the aid of the following dynamic data structures (an illustrative example is provided at the end of this section).

– **Step 1** For each vertex $i \in V_T$, remove $i$ from $V_T$ and remove the associated incidence edges from $E_T$, generally resulting in a number of disconnected components (step 1.1). Then, try to reconnect the components using edges between them (without leading to any cycle). The final solution after reconnection is typically a forest which is composed of a number of disconnected subtrees (step 1.2).

After that, temporarily store the roots (dynamically determined during the calculation, relying on the dynamic data structures slightly adapted from [25]) of all the subtrees into a one-dimensional array $A$ (step 1.3). Note that the length of $A$ (i.e., $|A|$) belongs to $[1, d_i]$, where $d_i$ is the degree of vertex $i$ in graph $G$.

**Complexity** We briefly discuss the computational complexity of step 1. At each iteration of local search, we just need to execute this step only once for each vertex $i \in V_T$, instead of running it repeatedly for each pair of vertices. Using dynamic updating techniques slightly adapted from the Steiner-vertex elimination move operator [25], at each iteration, the overall complexity needed to process all the vertices of $V_T$ (excluding step 1.3) could be bounded by $O(m \cdot \log n)$.

Given a forest corresponding to removing a vertex $i \in V_T$, one can obtain, with the aid of a union-find set data structure (dynamically updated), the root of each subtree within $O(d_{max})$ time, where $d_{max}$ is the maximum possible degree of each vertex in graph $G$. Then, after removing each vertex $i \in V_T$, we need to obtain the roots of at most $r_i + 1$ subtrees, where $r_i$ is the number of the children of vertex $i$ in solution $T$. This implies that at each iteration of local search, we should get the roots of at most $\sum_{i \in V_T} (r_i + 1) = \sum_{i \in V_T} r_i + |V_T| = |V_T| - 1 + |V_T| < 2n$ subtrees. Consequently, step 1.3 can be finished within an overall complexity of $O(n \cdot d_{max})$.

– **Step 2** For each vertex $j \notin V_T$ (for the sake of efficiency, here we only consider the vertices with at least one edge incident to solution $T$), examine all its adjacent vertices in $G$ and repeatedly check if a given adjacent vertex belongs to a subtree of the forest generated in step 1. If this is the case, store (temporarily) the root of the corresponding subtree into another one-dimensional array $B$. Notice that array $B$ may have duplicates, and the length of $B$ (i.e., $|B|$) belongs to $[0, d_j]$, where $d_j$ is the degree of vertex $j$ in graph $G$.

**Complexity** Unlike step 1, this step needs to be executed repeatedly for each pair of vertices, since the forests obtained in step 1 might be different corresponding to removing different vertices from $V_T$. As it is analyzed above, using union-find set (dynamically updated), getting the root of each subtree can be achieved within $O(d_{max})$, thus the complexity needed by step 2 for processing each pair of vertices [getting and storing the roots of at most $O(d_{max})$ subtrees] is bounded by $O(d_{max}^2)$.

– **Step 3** With the above two one-dimensional arrays $A$ (step 1) and $B$ (step 2), it is straightforward to verify the feasibility after swapping any pair of vertices. For this, it suffices to check if the root vertices stored in $B$ exactly cover the ones stored in $A$. If this is the case, the corresponding swap-vertex move is feasible, otherwise it is infeasible.

**Complexity** This step can be finished within a complexity of $O(|A| + |B|) \leq O(d_{max})$ for each pair of vertices.

To summarize, the overall complexity needed for verifying the feasibility of all the $O(n^2)$ possible swap-vertex moves is bounded by $O(m \cdot \log n + n^2 \cdot d_{max}^2)$, being affordable for mid-sized graphs [even for large-sized sparse graphs with $O(m) \approx O(n)$, generally corresponding to a small value of $d_{max}$].

Figure 1 is a SPG example showing how to verify feasibility after swapping any pair of vertices. Sub-figure (*a*) is the input graph with 25 vertices and 40 edges, where the 9 terminals are drawn in blocks and the 16 Steiner vertices are drawn in dots. For
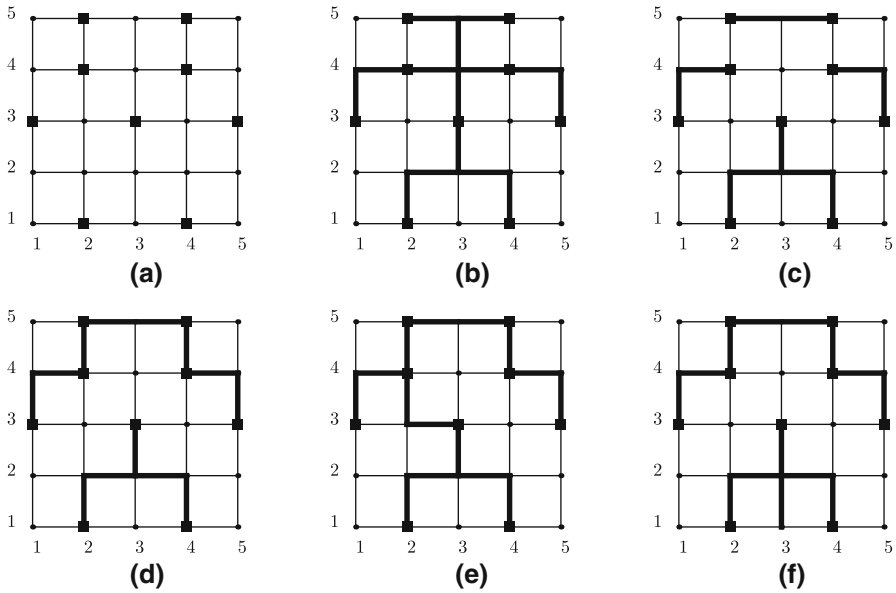
**Fig. 1** Example showing how to verify feasibility after swapping any pair of vertices

convenience, let $v_{xy}$ denote the vertex located at the cross-point of the $x$th horizontal line and the $y$th vertical line. Sub-figure ($b$) is a feasible solution spanning all the 9 terminals and 7 Steiner vertices (assume it is rooted at vertex $v_{53}$). Now, we consider for example the feasibility after swapping vertex $v_{43}$ with any other un-spanned vertex. For this, we at first delete vertex $v_{43}$ and the associated edges, leading to a forest (sub-figure $c$), which includes 4 subtrees rooted at $v_{53}$, $v_{42}$, $v_{44}$, $v_{33}$ respectively. After that, we try to reconnect these subtrees using edges between them. As shown in sub-figure ($d$), the subtrees rooted at $v_{53}$, $v_{42}$, $v_{44}$ could be merged into one subtree (rooted at $v_{53}$). Then the roots of the remaining two subtrees, i.e., $v_{53}$ and $v_{33}$, are stored into the one-dimensional array $A$.

With the above information, it is easy to verify the feasibility after swapping vertex $v_{43}$ with any other vertex. Consider vertex $v_{32}$ at first (similar for vertices $v_{34}$, $v_{21}$ and $v_{25}$), its adjacent vertices cover both subtrees rooted at $v_{53}$ and $v_{33}$, thus leading to a feasible solution (sub-figure $e$). By contrast, for vertex $v_{13}$, its adjacent vertices only cover one subtree (the subtree rooted at $v_{33}$), meaning that swapping $v_{43}$ and $v_{13}$ would lead to an infeasible solution (sub-figure $f$). This case is similar for vertices $v_{11}$, $v_{51}$, $v_{15}$ and $v_{55}$.

### 3.3 Evaluation function

Among all the feasible swap-vertex moves, there are generally a number of moves with the same move gain, which need to be further distinguished in order for the search to identify the most promising moves. For example, in a SPG instance with uniform edge costs, although swapping any two Steiner vertices would never change the objective

value (thus leads to the same zero move gain), it is still possible that some swap-vertex moves might be more promising than others in the sense that they lead to a solution which could be further improved by the search algorithm. However, the initial objective function alone cannot allow us to identify the most promising swap-vertex moves. In order to overcome this difficulty, we design a more discriminating evaluation function which aims to guide the search towards promising solutions, inspired by our previous experiences on graph labeling problems [33,34].

Let us consider a SPG instance with uniform edge costs at first. In this case, feasibly removing a Steiner vertex would definitely improve the objective value. Now, let $T$ be a solution where no Steiner vertex could be feasibly removed. Although swapping any Steiner vertex in $T$ and another vertex out of $T$ would never change the objective value, it may become possible again to feasibly remove some Steiner vertex, thus further improving the objective value (see the example of Fig. 2 and the explanations given at the end of this section). This observation could be similarly extended to the PCSPG and the MWCS. Respectively, in any PCSPG instance with uniform edge costs, feasibly removing a vertex with prize lower than the cost of each edge would certainly improve the objective value. In any MWCS instance, feasibly removing a vertex with negative weight would definitely lead to an improving solution (regardless of the edge costs).

Inspired by these observations, for the three STP variants studied in this paper, we define an auxiliary evaluation function as follows, which aims to estimate the opportunity of feasibly removing vertices.

**Definition 1** 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, we observe that if $sd_i = 1$, it means vertex $i$ is directly reachable from only one other vertex $j \in V_T$, implying that once the related vertex $j$ is removed from $V_T$, the resulting solution would become disconnected (unless $|V_T| = 2$).

**Definition 2** The *special degree* $sd(T)$ of a feasible solution $T$ is defined as the number of vertices with special degree $sd_i = 1$, i.e.:

$$sd(T) = \sum_{i \in V_T, sd_i=1} 1. \tag{7}$$

The values of $sd(T)$ could be dynamically evaluated, within a complexity of $O(d_{max})$ after swapping each pair of vertices.

Intuitively, the lower the value of $sd(T)$, the larger the opportunity to feasibly remove a vertex (thus the larger the opportunity to further improve the objective value). For each of the three STP variants studied in this paper, to identify improving solutions,
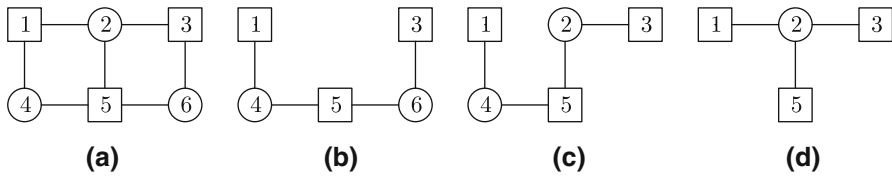
**Fig. 2** Example of applying the swap-vertex move guided by the auxiliary evaluation function

we use the objective value defined in Sect. 2 as the main evaluation criterion, and adopt the special degree $sd(T)$ as an auxiliary evaluation criterion. Precisely, we say solution $T1$ dominates solution $T2$ if (1) the objective value of $T1$ is better than that of $T2$, or (2) $T1$ and $T2$ have the same objective value, but $T1$ has a special degree lower than $T2$.

Figure 2 is a SPG example (with $c_e = 1$, $\forall e \in E$) which shows the search process of applying the swap-vertex move operator, guided by the auxiliary evaluation function. Respectively, sub-figure ($a$) is the input graph with three terminals (vertex 1, 3, 5, drawn in boxes) and three Steiner vertices (vertex 2, 4, 6, drawn in circles). Sub-figure ($b$) is an initial solution with objective value $f(b) = 4$. Notice that solution ($b$) could not be further improved by simply adding or removing any Steiner vertex. However, if we swap vertex 6 with vertex 2 to get a new solution ($c$), although the objective value does not vary, the special degree decreases. More precisely, $sd(b) = 2$ (both vertex 1 and 3 have a special degree $sd_1 = sd_3 = 1$), and $sd(c) = 1$ (only vertex 3 has a special degree $sd_3 = 1$). Now, from solution ($c$), we could feasibly remove Steiner vertex 4, to obtain an improving solution ($d$) with objective value $f(d) = 3$. Obviously, the swap-vertex move operator as well as the auxiliary evaluation function play a crucial role during this search process.

## 4 Experimental results

This section is dedicated to a computational assessment of the proposed swap-vertex move operator as well as the associated auxiliary evaluation function. This study is based on a set of representative benchmark instances for each of the three studied STP variants (SPG, PCSPG and MWCS).

### 4.1 Experimental protocol

In order to particularly emphasize the importance of the swap-vertex move operator as well as the auxiliary evaluation function, we avoid to test them within a complex algo-rithm. Instead, we implement and compare three simple local search procedures where the search is fully driven by the employed neighborhoods and evaluation functions. We respectively call these three procedures 'Basic-LS' (with add and remove moves of Sect. 3.1, but without the swap-vertex move described in Sect. 3.1), 'Swap-LS' (Basic-LS reinforced by the swap-vertex move, but without the auxiliary evaluation function defined in Sect. 3.3), and 'Enhanced-LS' (Basic-LS reinforced by the swap-vertex move and the auxiliary evaluation function). All these three procedures were

coded in the C language,[2] shared the same data structures whenever it is possible and were executed on the same computing platform, i.e., a machine with an Intel(R) Core(TM) i5-4460 3.20GHz processor with 4 cores and 4GB RAM. Note that each job occupied only one core, being executed in sequential order.

---

**Algorithm 1** Procedure of Basic-LS

---

1: **Input:** A randomly generated initial solution $T$
2: **Output:** A local optimal solution
3: Let neighborhood $N(T) \leftarrow N_1(T) \cup N_2(T)$ /*defined in Section 3.1*/
4: **for** Each neighboring solution $T^* \in N(T)$ (examined in random order) **do**
5:   **if** $T^*$ has a better objective value (defined in Section 2) than $T$ **then**
6:     Let $T \leftarrow T^*$ and go to line 3
7:   **end if**
8: **end for**
9: **return** $T$

---

**Algorithm 2** Procedure of Swap-LS

---

1: **Input:** A randomly generated initial solution $T$
2: **Output:** A local optimal solution
3: Let neighborhood $N(T) \leftarrow N_1(T) \cup N_2(T) \cup N_3(T)$ /*defined in Section 3.1*/
4: **for** Each neighboring solution $T^* \in N(T)$ (examined in random order) **do**
5:   **if** $T^*$ has a better objective value (defined in Section 2) than $T$ **then**
6:     Let $T \leftarrow T^*$ and go to line 3
7:   **end if**
8: **end for**
9: **return** $T$

---

**Algorithm 3** Procedure of Enhanced-LS

---

1: **Input:** A randomly generated initial solution $T$
2: **Output:** A local optimal solution
3: Let neighborhood $N(T) \leftarrow N_1(T) \cup N_2(T) \cup N_3(T)$ /*defined in Section 3.1*/
4: **for** Each neighboring solution $T^* \in N(T)$ (examined in random order) **do**
5:   **if** $T^*$ has a objective value better than $T$ or their objective values are the same but $T^*$ has a special degree (defined in Section 3.3) lower then $T$ **then**
6:     Let $T \leftarrow T^*$ and go to line 3
7:   **end if**
8: **end for**
9: **return** $T$

---

– **Basic-LS** (Algorithm 1): Starting from a given initial solution $T$, Basic-LS examines in random order the solutions of $N_1(T) \cup N_2(T)$, i.e., the union of the two neighborhoods induced by the add and remove operators (defined in Sect. 3.1, using

---

2 Our source codes are available at: http://www.info.univ-angers.fr/pub/hao/SPG_SWAP/CODE.zip.

the dynamic updating techniques described in [25] for efficient evaluations), and iteratively replaces the incumbent solution with the first met improving solution (only regarding the objective value). The process stops when no improving solution exists in the whole neighborhood, meaning that a local optimum is reached.

– **Swap-LS** (Algorithm 2): Swap-LS is an improved version of Basic-LS. It realizes the search with an enlarged neighborhood $N_1(T) \cup N_2(T) \cup N_3(T)$, i.e., the union of the three neighborhoods induced by the add, remove and swap-vertex operators (defined in Sect. 3.1).

– **Enhanced-LS** (Algorithm 3): Enhanced-LS is the same as Swap-LS with the following difference. Enhanced-LS uses, instead of the original objective function, the auxiliary evaluation function described in Sect. 3.3 to evaluate the perspective of each neighboring solution.

We did not implement and compare a variant with the conventional move operators (add and remove) combined with the auxiliary evaluation function. In fact, without the swap-vertex move operator, the auxiliary evaluation function is of little interest since in general adding or removing a vertex will change the objective value.

Additionally, the above three algorithms use the following heuristic for generating initial solutions (uniform for all the three studied STP variants). We call a terminal vertex (for the SPG), or a vertex with positive prize (for the PCSPG) or positive weight (for the MWCS) as a candidate vertex. Then, starting from a randomly selected candidate vertex, we try to complete the incumbent solution by inserting a new path connecting a randomly chosen un-spanned candidate vertex (for the PCSPG and the MWCS, we should guarantee that the objective value after insertion is improved). This process is repeated until no further path can be inserted, meaning that an initial solution is generated.

In the following subsections, we compare the performances of the three algorithms on the SPG, the PCSPG and the MWCS, using a set of representative benchmarks for each problem. To ensure fair comparisons, for each test instance, we repeatedly ran each algorithm, each run independently restarting from a randomly generated initial solution. This process was repeated until one CPU hour was reached.

Finally, we also briefly summarize the competition results of the 11th DIMACS implementation challenge, in order to provide some supplementary information about the effectiveness of the techniques introduced in this paper.

## 4.2 Results on the SPG

We chose 25 SPG instances (the un-weighted instances of types hc, bip and cc) from the 11th DIMACS challenge as benchmarks to evaluate the performances of the three algorithms. These instances are all with uniform edge costs and of reasonable size ($|V| < 6000$), thus being suitable to apply the swap-vertex move operator. Moreover, they are known to be extremely difficult for the existing approaches such that most of these instances still remain open.

The detailed results are summarized in Table 1, where the first two columns indicate the instance name and the corresponding optimal result (extracted from [20], if

**Table 1** Comparative results between Basic-LS, Swap-LS and Enhanced-LS on 25 representative SPG instances (within one CPU hour)

| Instance | Optimal | Basic-LS | | Swap-LS | | | Enhanced-LS | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $f^{best}$ | $f^{avg}$ | $f^{best}$ | $f^{avg}$ | Improve (%) | $f^{best}$ | $f^{avg}$ | Improve (%) |
| hc6u.stp | 39.00 | 39.00* | 43.80 | 39.00* | 43.79 | 0.00 | 39.00* | 39.28 | 0.00 |
| hc7u.stp | 77.00 | 77.00* | 87.63 | 77.00* | 87.63 | 0.00 | 77.00* | 77.30 | 0.00 |
| hc8u.stp | 148.00 | 153.00 | 173.86 | 153.00 | 173.89 | 0.00 | 148.00* | 150.84 | 3.27 |
| hc9u.stp | – | 304.00 | 345.92 | 307.00 | 345.84 | −0.99 | 293.00 | 298.37 | 3.62 |
| hc10u.stp | – | 624.00 | 687.27 | 623.00 | 687.21 | 0.16 | 586.00 | 591.30 | 6.09 |
| hc11u.stp | – | 1268.00 | 1366.94 | 1270.00 | 1366.88 | −0.16 | 1169.00 | 1173.97 | 7.81 |
| hc12u.stp | – | 2593.00 | 2720.42 | 2569.00 | 2721.86 | 0.93 | 2321.00 | 2333.92 | 10.49 |
| bip42u.stp | 236.00 | 258.00 | 264.02 | 258.00 | 264.02 | 0.00 | 239.00 | 243.48 | 7.36 |
| bip52u.stp | – | 263.00 | 268.05 | 264.00 | 268.05 | −0.38 | 238.00 | 241.22 | 9.51 |
| bip62u.stp | – | 235.00 | 239.83 | 234.00 | 239.83 | 0.43 | 221.00 | 223.87 | 5.96 |
| bipa2u.stp | – | 374.00 | 377.72 | 374.00 | 377.72 | 0.00 | 343.00 | 347.35 | 8.29 |
| bipe2u.stp | 54.00 | 55.00 | 57.00 | 55.00 | 57.00 | 0.00 | 54.00* | 54.88 | 1.82 |
| cc3-4n.stp | 13.00 | 13.00* | 13.40 | 13.00* | 13.40 | 0.00 | 13.00* | 13.00 | 0.00 |
| cc3-5n.stp | 20.00 | 20.00* | 21.77 | 20.00* | 21.77 | 0.00 | 20.00* | 20.07 | 0.00 |
| cc3-10n.stp | – | 76.00 | 89.08 | 76.00 | 89.08 | 0.00 | 75.00 | 75.57 | 1.32 |
| cc3-11n.stp | – | 93.00 | 109.25 | 94.00 | 109.25 | −1.08 | 92.00 | 92.32 | 1.08 |
| cc3-12n.stp | – | 113.00 | 133.47 | 115.00 | 133.47 | −1.77 | 111.00 | 111.70 | 1.77 |
| cc5-3n.stp | 42.00 | 44.00 | 49.20 | 44.00 | 49.20 | 0.00 | 43.00 | 46.17 | 2.27 |
| cc6-2n.stp | 18.00 | 18.00* | 19.86 | 18.00* | 19.86 | 0.00 | 18.00* | 18.00 | 0.00 |
| cc6-3n.stp | 100.00 | 106.00 | 114.88 | 106.00 | 114.88 | 0.00 | 104.00 | 109.08 | 1.89 |

**Table 1** continued

| Instance | Optimal | Basic-LS | | Swap-LS | | | Enhanced-LS | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $f^{best}$ | $f^{avg}$ | $f^{best}$ | $f^{avg}$ | Improve (%) | $f^{best}$ | $f^{avg}$ | Improve (%) |
| cc7-3n.stp | – | 318.00 | 332.67 | 318.00 | 332.67 | 0.00 | **298.00** | **305.05** | 6.29 |
| cc9-2n.stp | – | 106.00 | 113.80 | 106.00 | **113.79** | 0.00 | **102.00** | **107.26** | 3.77 |
| cc10-2n.stp | – | 194.00 | 205.36 | 195.00 | 205.36 | −0.52 | **185.00** | **189.96** | 4.64 |
| cc11-2n.stp | – | 358.00 | 372.90 | 358.00 | 372.90 | 0.00 | **340.00** | **348.71** | 5.03 |
| cc12-2n.stp | – | 679.00 | 696.14 | **675.00** | **696.09** | 0.59 | **639.00** | **645.89** | 5.89 |
| Average | – | – | – | – | – | −0.11% | – | – | 3.93% |

applicable). The next two columns respectively indicate the overall best objective value found by Basic-LS (indicated with a symbol "*" if it reaches the known optimal result), and the average objective value of each independent run. For comparisons, columns 5 and 6 indicate the same information corresponding to Swap-LS, where the objective values better than those of Basic-LS are indicated in bold, and the same objective values are indicated in italic. Additionally, column 7 gives the mean improvement percentage obtained by Swap-LS compared to Basic-LS, in terms of best objective values. Finally, we give in the last three columns similar information corresponding to Enhanced-LS.

In terms of best objective values, compared to Basic-LS, Swap-LS obtains within the same allowed time (one CPU hour for each instance) 4 better, 6 worse and 15 equal results, corresponding to a mean deterioration of 0.11% (averaged on all these 25 instances) provided by Swap-LS. Under the same test condition, compared to Basic-LS, Enhanced-LS obtains 20 better results and yields the same results on the remaining 5 instances (all reaching optimality), leading to a mean improvement of 3.93%. Furthermore, we used the Friedman test to check the statistical differences between the best objective values of the compared algorithms, which respectively reveals a $p$ value of $9.39 \times 10^{-1}$ (between Swap-LS and Basic-LS), a $p$ value of $3.75 \times 10^{-6}$ (between Enhanced-LS and Basic-LS), and a $p$ value of $3.57 \times 10^{-7}$ (between Enhanced-LS and Swap-LS). These results indicate that, in terms of best objective values, Enhanced-LS performs clearly better than Basic-LS and Swap-LS, while there is no significant difference between Swap-LS and Basic-LS.

In terms of average objective values, compared to Basic-LS, Swap-LS obtains 6 better, 2 worse and 17 equal results, while Enhanced-LS succeeds to obtain a better result on each of the 25 test instances. The Friedman test in terms of average objective values respectively reveals a $p$ value of $8.09 \times 10^{-1}$ (between Swap-LS and Basic-LS), a $p$ value of $3.69 \times 10^{-9}$ (between Enhanced-LS and Basic-LS), and a $p$ value of $9.07 \times 10^{-8}$ (between Enhanced-LS and Swap-LS), indicating again that Enhanced-LS is a much improved version over Basic-LS and Swap-LS, while Swap-LS does not perform significantly differently from Basic-LS.

To conclude, the swap-vertex move operator alone does not perform well enough on the SPG instances with uniform edge costs (on these instances swapping any two Steiner vertices would never change the objective value). In contrast, when it is combined with the auxiliary evaluation function, we obtain a highly competitive algorithm to solve the SPG instances with uniform edge costs.

## 4.3 Results on the PCSPG

For the PCSPG, we chose 40 benchmark instances with uniform edge costs from the 11th DIMACS challenge, including the un-weighted instances of groups H, H2, and all the instances of groups PUCNU, ACTMODPC (we do not use the HAND instances since they are too large to apply the swap-vertex move operator). These instances are also very challenging since many of them still remain unsolved. The corresponding optimal results are also extracted from [20] (if known). Like before, for each instance,

we repeatedly ran Basic-LS (respectively, Swap-LS and Enhanced-LS) for one CPU hour, and show the obtained results in Table 2, with the same information as in Table 1.

Table 2 discloses that, in terms of best objective values, Swap-LS obtains within the same allowed time 8 better, 3 worse and 29 equal results compared to Basic-LS, corresponding to a mean improvement of 0.03%. For comparison, Enhanced-LS obtains 28 better and 12 equal results compared to Basic-LS, leading to a mean improvement of 2.11%. Furthermore, the Friedman test in terms of best objective values reveals a $p$ value of $5.60 \times 10^{-1}$ (between Swap-LS and Basic-LS), a $p$ value of $2.71 \times 10^{-9}$ (between Enhanced-LS and Basic-LS), and a $p$ value of $6.09 \times 10^{-7}$ (between Enhanced-LS and Swap-LS), indicating again that Enhanced-LS performs clearly better than Basic-LS and Swap-LS, while there is no significant difference between Swap-LS and Basic-LS.

In terms of average objective values, compared to Basic-LS, Swap-LS obtains 22 better, 8 worse and 10 equal results. In contrast, Enhanced-LS obtains a better average result on 39 out of the 40 test instances, only with one exception on which Enhanced-LS yields a same average result. The Friedman test in terms of average objective values respectively reveals a $p$ value of $1.11 \times 10^{-1}$ (between Swap-LS and Basic-LS), a $p$ value of $1.73 \times 10^{-12}$ (between Enhanced-LS and Basic-LS), and a $p$ value of $7.02 \times 10^{-7}$ (between Enhanced-LS and Swap-LS), indicating that Swap-LS performs slightly better than Basic-LS (especially on the last eight instances of group ACTMODPC with quite different structures), while Enhanced-LS performs overall clearly better than both Basic-LS and Swap-LS.

This analysis showed that on most of the PCSPG instances with uniform edge costs, it is very useful to combine the swap-vertex move operator with the auxiliary evaluation function within a local-search based algorithm.

### 4.4 Results on the MWCS

Now we turn our attention to the more general MWCS, for which any instance of reasonable size is suitable to apply the swap-vertex move operator (there are no edge costs for the MWCS). More importantly, as explained in Sect. 2, the MWCS is a basic model which potentially covers many related STP variants (including both the SPG and the PCSPG), indicating the generality of the swap-vertex move operator for tackling various STP variants.

We chose as benchmarks the 32 representative MWCS instances which were adopted as the final test set by the 11th DIMACS challenge. Similarly, for each of these 32 instances, we repeatedly and independently ran Basic-LS, Swap-LS and Enhanced-LS with a cutoff limit of one CPU hour. The obtained results are listed in Table 3, where we display the same statistics as in the previous tables. As indicated in column 2, all these instances have known optimal results [20,23].

As shown in Table 3, in terms of best objective values, within the same allowed time, Swap-LS obtains 19 better, 1 worse and 12 equal results compared to Basic-LS, corresponding to a mean improvement of 2.09%. Under the same test condition, Enhanced-LS also obtains 19 better, 1 worse and 12 equal results compared to Basic-LS, leading to a mean improvement of 2.20%. Respectively, the Friedman test in

**Table 2** Comparative results between Basic-LS, Swap-LS and Enhanced-LS on 40 representative PCSPG instances (within one CPU hour)

| Instance | Optimal | Basic-LS | | Swap-LS | | | Enhanced-LS | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | $f^{best}$ | $f^{avg}$ | $f^{best}$ | $f^{avg}$ | Improve (%) | $f^{best}$ | $f^{avg}$ | Improve (%) |
| hc6u.stp | 36.00 | 36.00* | 38.77 | 36.00* | 38.77 | 0.00 | 36.00* | 37.04 | 0.00 |
| hc7u.stp | 72.00 | 73.00 | 77.62 | 73.00 | 77.61 | 0.00 | 72.00* | 73.60 | 1.37 |
| hc8u.stp | 143.00 | 146.00 | 154.14 | 146.00 | 154.15 | 0.00 | 143.00* | 145.83 | 2.05 |
| hc9u.stp | 283.00 | 295.00 | 310.53 | 295.00 | 310.51 | 0.00 | 284.00 | 289.56 | 3.73 |
| hc10u.stp | – | 588.00 | 609.66 | 588.00 | 609.64 | 0.00 | 565.00 | 571.15 | 3.91 |
| hc11u.stp | – | 1184.00 | 1216.96 | 1178.00 | 1216.87 | 0.51 | 1131.00 | 1140.03 | 4.48 |
| hc12u.stp | – | 2381.00 | 2426.55 | 2378.00 | 2427.14 | 0.13 | 2255.00 | 2266.11 | 5.29 |
| hc6u2.stp | 20.00 | 20.00* | 21.54 | 20.00* | 21.55 | 0.00 | 20.00* | 20.12 | 0.00 |
| hc7u2.stp | 47.00 | 47.00* | 51.16 | 47.00* | 51.16 | 0.00 | 47.00* | 47.66 | 0.00 |
| hc8u2.stp | 97.00 | 99.00 | 104.36 | 98.00 | 104.35 | 1.01 | 97.00* | 99.39 | 2.02 |
| hc9u2.stp | – | 196.00 | 204.81 | 196.00 | 204.79 | 0.00 | 190.00 | 193.84 | 3.06 |
| hc10u2.stp | – | 398.00 | 414.90 | 398.00 | 414.93 | 0.00 | 383.00 | 387.66 | 3.77 |
| hc11u2.stp | – | 799.00 | 818.22 | 799.00 | 818.21 | 0.00 | 761.00 | 768.20 | 4.76 |
| hc12u2.stp | – | 1602.00 | 1631.39 | 1602.00 | 1631.64 | 0.00 | 1516.00 | 1524.11 | 5.37 |
| bip42nu.stp | 226.00 | 244.00 | 251.08 | 244.00 | 251.08 | 0.00 | 228.00 | 231.59 | 6.56 |
| bip52nu.stp | 222.00 | 246.00 | 250.07 | 246.00 | 250.06 | 0.00 | 224.00 | 226.66 | 8.94 |
| bip62nu.stp | 214.00 | 227.00 | 232.26 | 227.00 | 232.26 | 0.00 | 215.00 | 217.91 | 5.29 |
| bipa2nu.stp | – | 358.00 | 363.63 | 357.00 | 363.64 | 0.28 | 327.00 | 330.76 | 8.66 |
| bipe2nu.stp | 53.00 | 54.00 | 55.04 | 54.00 | 55.05 | 0.00 | 53.00* | 53.18 | 1.85 |
| cc3-4nu.stp | 10.00 | 10.00* | 10.90 | 10.00* | 10.90 | 0.00 | 10.00* | 10.59 | 0.00 |
| cc3-5nu.stp | 17.00 | 17.00* | 18.52 | 17.00* | 18.52 | 0.00 | 17.00* | 17.96 | 0.00 |
| cc3-10nu.stp | – | 62.00 | 65.63 | 62.00 | 65.62 | 0.00 | 62.00 | 63.77 | 0.00 |

**Table 2** continued

| Instance | Optimal | Basic-LS | | Swap-LS | | | Enhanced-LS | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $f^{best}$ | $f^{avg}$ | $f^{best}$ | $f^{avg}$ | Improve (%) | $f^{best}$ | $f^{avg}$ | Improve (%) |
| cc3-11nu.stp | – | 86.00 | 89.96 | 86.00 | 89.96 | 0.00 | **85.00** | **86.55** | 1.16 |
| cc3-12nu.stp | – | 99.00 | 107.45 | 99.00 | **107.45** | 0.00 | **98.00** | **103.34** | 1.01 |
| cc5-3nu.stp | 36.00 | 37.00 | 38.29 | 37.00 | 38.29 | 0.00 | **37.00** | **38.23** | 0.00 |
| cc6-2nu.stp | 15.00 | 15.00* | 15.23 | 15.00* | 15.23 | 0.00 | 15.00* | 15.23 | 0.00 |
| cc6-3nu.stp | 95.00 | 98.00 | 104.48 | 98.00 | 104.47 | 0.00 | **97.00** | **103.46** | 1.02 |
| cc7-3nu.stp | – | 288.00 | 298.27 | 288.00 | 298.30 | 0.00 | **279.00** | **287.80** | 3.13 |
| cc9-2nu.stp | 83.00 | 85.00 | 89.20 | 85.00 | 89.20 | 0.00 | **84.00** | **88.91** | 1.18 |
| cc10-2nu.stp | – | 173.00 | 182.17 | 174.00 | 182.15 | −0.58 | **172.00** | **179.92** | 0.58 |
| cc11-2nu.stp | – | 320.00 | 332.89 | 321.00 | 332.87 | −0.31 | **312.00** | **325.05** | 2.50 |
| cc12-2nu.stp | – | 597.00 | 613.06 | 598.00 | 612.68 | −0.17 | **583.00** | **597.81** | 2.35 |
| HCMV.stp | 7371.54 | 7373.39 | 7382.61 | 7373.39 | 7379.61 | 0.00 | 7373.39 | 7379.61 | 0.00 |
| lymphoma.stp | 3341.89 | 3371.24 | 3412.78 | 3371.24 | 3406.95 | 0.00 | 3371.24 | 3406.96 | 0.00 |
| metabol_expr_mice_1.stp | 11,346.93 | 11,485.55 | 11,838.56 | **11,485.06** | **11,834.59** | 0.00 | **11,485.06** | **11,835.06** | 0.00 |
| metabol_expr_mice_2.stp | 16,250.24 | 16,342.56 | 16,495.46 | 16,342.56 | **16,493.85** | 0.00 | 16,342.56 | 16,493.82 | 0.00 |
| metabol_expr_mice_3.stp | 16,919.62 | 17,173.26 | 17,408.79 | **17,159.42** | **17,406.32** | 0.08 | **17,159.42** | **17,406.29** | 0.08 |
| drosophila001.stp | 8273.98 | 8286.43 | 8300.51 | 8286.43 | **8297.74** | 0.00 | 8286.43 | **8297.74** | 0.00 |
| drosophila005.stp | 8121.31 | 8211.42 | 8268.89 | **8197.07** | **8261.44** | 0.17 | **8195.64** | 8261.72 | 0.19 |
| drosophila0075.stp | 8039.86 | 8133.80 | 8210.47 | **8120.13** | 8202.67 | 0.17 | **8122.07** | 8201.17 | 0.14 |
| Average | – | – | – | – | – | 0.03% | – | – | 2.11% |

**Table 3** Comparative results between Basic-LS, Swap-LS and Enhanced-LS on 32 representative MWCS instances (within one CPU hour)

| Instance | Optimal | Basic-LS | | Swap-LS | | | Enhanced-LS | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $f^{best}$ | $f^{avg}$ | $f^{best}$ | $f^{avg}$ | Improve (%) | $f^{best}$ | $f^{avg}$ | Improve (%) |
| drosophila001.stp | 24.4 | 11.9 | 3.27 | 11.9 | 3.31 | 0.00 | 11.9 | 3.31 | 0.00 |
| drosophila005.stp | 178.7 | 150.3 | 109.02 | 165.3 | 120.52 | 9.93 | 165.3 | 119.59 | 9.97 |
| drosophila0075.stp | 260.5 | 231.8 | 211.88 | 246.4 | 222.25 | 6.29 | 243.1 | 227.04 | 4.87 |
| HCMV.stp | 7.6 | 5.0 | 0.64 | 5.0 | 0.64 | 0.00 | 5.0 | 0.64 | 0.00 |
| lymphoma.stp | 70.2 | 61.1 | 31.55 | 60.9 | 32.72 | −0.39 | 60.9 | 32.69 | −0.39 |
| metabol_expr_mice_1.stp | 544.9 | 533.1 | 411.17 | 538.6 | 418.53 | 1.02 | 537.5 | 421.65 | 0.82 |
| metabol_expr_mice_2.stp | 241.1 | 241.1* | 114.44 | 241.1* | 116.46 | 0.00 | 241.1* | 116.49 | 0.00 |
| metabol_expr_mice_3.stp | 508.3 | 477.3 | 308.11 | 495.8 | 320.12 | 3.88 | 495.8 | 319.77 | 3.88 |
| 1500-a-0_6-d-0_25-e-0_25.stp | 1333.5 | 1193.6 | 1118.52 | 1255.5 | 1205.98 | 5.18 | 1261.3 | 1218.59 | 5.66 |
| 1500-a-0_6-d-0_25-e-0_5.stp | 2799.7 | 2787.2 | 2748.63 | 2799.7* | 2788.98 | 0.45 | 2799.7* | 2792.24 | 0.45 |
| 750-a-0_647-d-0_25-e-0_25.stp | 702.6 | 672.9 | 621.08 | 701.4 | 677.79 | 4.24 | 701.8 | 682.00 | 4.29 |
| 1500-a-1-d-0_25-e-0_25.stp | 1377.0 | 1376.6 | 1362.93 | 1377.0* | 1368.65 | 0.03 | 1377.0* | 1368.56 | 0.03 |
| 1000-a-0_6-d-0_5-e-0_25.stp | 522.5 | 439.7 | 361.10 | 501.8 | 435.81 | 14.11 | 514.8 | 449.79 | 17.08 |
| 1000-a-0_6-d-0_25-e-0_5.stp | 1872.3 | 1866.3 | 1847.82 | 1872.3* | 1863.33 | 0.32 | 1872.3* | 1863.36 | 0.32 |
| 1000-a-0_6-d-0_25-e-0_25.stp | 931.5 | 875.6 | 800.01 | 925.1 | 886.88 | 5.65 | 922.2 | 892.35 | 5.33 |
| 750-a-0_647-d-0_25-e-0_5.stp | 1419.8 | 1419.4 | 1398.21 | 1419.8* | 1415.41 | 0.03 | 1419.8* | 1416.14 | 0.03 |
| 500-a-0_62-d-0_75-e-0_25.stp | 171.6 | 170.3 | 147.76 | 171.6* | 165.63 | 0.81 | 171.6* | 165.80 | 0.81 |
| 500-a-0_62-d-0_25-e-0_25.stp | 460.6 | 432.9 | 387.02 | 449.9 | 430.15 | 3.92 | 448.7 | 431.71 | 3.64 |

**Table 3** continued

| Instance | Optimal | Basic-LS | | Swap-LS | | | Enhanced-LS | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $f^{best}$ | $f^{avg}$ | $f^{best}$ | $f^{avg}$ | Improve (%) | $f^{best}$ | $f^{avg}$ | Improve (%) |
| 1000-a-0_6-d-0_75-e-0_25.stp | 332.8 | 324.2 | 289.04 | **332.5** | **309.49** | 2.57 | **332.8***  | **313.28** | 2.66 |
| 500-a-0_62-d-0_25-e-0_5.stp | 993.0 | 986.4 | 961.43 | **993.0*** | **982.66** | 0.66 | **993.0*** | **983.59** | 0.66 |
| 1000-a-1-d-0_25-e-0_75.stp | 2789.6 | 2789.6* | 2789.57 | 2789.6* | **2789.58** | 0.00 | 2789.6* | **2789.58** | 0.00 |
| 1000-a-0_6-d-0_25-e-0_75.stp | 2789.6 | 2789.6* | 2787.48 | 2789.6* | **2789.58** | 0.00 | 2789.6* | **2789.56** | 0.00 |
| 1500-a-0_6-d-0_75-e-0_75.stp | 1423.6 | 1417.2 | 1388.62 | **1423.6*** | **1411.81** | 0.45 | **1423.6*** | **1419.14** | 0.45 |
| 1000-a-1-d-0_5-e-0_75.stp | 1770.3 | 1770.3* | 1770.28 | 1770.3* | 1770.28 | 0.00 | 1770.3* | 1770.28 | 0.00 |
| 1000-a-0_6-d-0_5-e-0_75.stp | 1762.7 | 1762.7* | 1753.70 | 1762.7* | **1757.67** | 0.00 | 1762.7* | **1757.59** | 0.00 |
| 1500-a-1-d-0_75-e-0_75.stp | 1423.6 | 1423.6* | 1423.55 | 1423.6* | **1423.61** | 0.00 | 1423.6* | **1423.61** | 0.00 |
| 1500-a-1-d-0_75-e-0_5.stp | 1089.8 | 1089.8* | 1084.94 | 1089.8* | **1089.31** | 0.00 | 1089.8* | **1089.50** | 0.00 |
| 1000-a-1-d-0_25-e-0_5.stp | 1883.2 | 1883.2* | 1881.13 | 1883.2* | **1883.21** | 0.00 | 1883.2* | **1883.21** | 0.00 |
| 1500-a-0_6-d-0_5-e-0_25.stp | 847.5 | 737.0 | 651.80 | 782.4 | 728.18 | 6.16 | 797.2 | 745.30 | 8.17 |
| 1500-a-0_6-d-0_75-e-0_5.stp | 1089.8 | 1073.5 | 1033.04 | **1089.8*** | **1074.27** | 1.52 | **1089.8*** | **1081.21** | 1.52 |
| 750-a-1-d-0_25-e-0_75.stp | 2116.6 | 2116.6* | 2116.58 | 2116.6* | **2116.58** | 0.00 | 2116.6* | 2116.58 | 0.00 |
| 750-a-0_647-d-0_25-e-0_75.stp | 2116.6 | 2116.6* | 2114.61 | 2116.6* | **2116.58** | 0.00 | 2116.6* | **2116.58** | 0.00 |
| Average | – | – | – | – | – | 2.09% | – | – | 2.20% |

terms of best objective values reveals a $p$ value of $1.75 \times 10^{-5}$ (between Swap-LS and Basic-LS), a $p$ value of $7.77 \times 10^{-6}$ (between Enhanced-LS and Basic-LS) and a $p$ value of $9.84 \times 10^{-1}$ (between Enhanced-LS and Swap-LS), indicating that there is no significant difference between Enhanced-LS and Swap-LS, while they both clearly dominate Basic-LS.

In terms of average objective values, compared to Basic-LS, Swap-LS and Enhanced-LS both obtains 29 better and 3 equal results, while the Friedman test respectively reveals a $p$ value of $1.70 \times 10^{-6}$ (between Swap-LS and Basic-LS), a $p$ value of $4.10 \times 10^{-10}$ (between Enhanced-LS and Basic-LS) and a $p$ value of $3.46 \times 10^{-1}$ (between Enhanced-LS and Swap-LS), indicating again that Enhanced-LS and Swap-LS performs similarly, while they both perform clearly better than Basic-LS.

To conclude, on these 32 representative MWCS instances, the swap-vertex move operator alone contributes significantly to the effectiveness of the proposed algorithm, while the auxiliary evaluation function is not so important as for the SPG and the PCSPG instances. One possible reason is that on these MWCS instances, different vertices generally have different weights, thus swapping a pair of vertices rarely leads to a neighboring solution with a $\Delta = 0$, making the auxiliary evaluation function irrelevant (actually, this is also the case for the last 8 PCSPG instances of Table 2).

### 4.5 Performance on the 11th DIMACS Implementation Challenge

By combining the techniques described in this work with several other strategies (e.g., tabu search, knowledge learning mechanisms, and adaptive perturbation strategies), we proposed KTS, a knowledge guided tabu search algorithm [28] for the 11th DIMACS implementation challenge dedicated to the broadly defined Steiner tree problems [31]. We took part in the competition on two STP variants, i.e., the RPCST and the PCSPG respectively, which attracted the largest number of participants among the seven competition variants. For each variant, we participated in four main challenge subcategories involving all the competing algorithms, i.e., two *Primal Integral* subcategories regarding both the solution quality and runtime, and two *Primal Bound* subcategories regarding only the solution quality. Given that our KTS algorithm is a heuristic which runs only in sequential mode, we are not concerned by the special subcategories focused on exact algorithms or parallelized algorithms.

Under the competition rules, the KTS algorithm achieved the following performances on the eight subcategories in which it was involved.[3] On the RPCST, it won both the two *Primal Integral* subcategories, and was tied for the first place on both the two *Primal Bound* subcategories. On the PCSPG, it was ranked first on one *Primal Integral* subcategory, and third on the remaining three subcategories. Additionally, although we did not participate in the competition on the SPG (due to time reason), the KTS algorithm yielded remarkable results on several challenging SPG instances, with respect to the current best known results.

Actually, as key elements of the KTS algorithm, the swap-vertex move operator and the auxiliary evaluation function contributed significantly to the achieved perfor-

---

[3] Details at: http://dimacs11.zib.de/contest/results/results.html.

mance on the 12 difficult PCSPG instances of groups H, PUCNU and ACTMODPC, including 10 instances with strictly uniform edge costs and 2 instances with nearly uniform edge costs. When these two components are disabled from the KTS algorithm, the performance decreases drastically on these 12 instances. Given the applicability condition of the swap-vertex move operator, it was not applied by KTS to solve the remaining challenge instances with significantly varying edge costs or of too large size (with more than 6000 vertices).

## 5 Conclusions

In this paper, we developed an effective swap-vertex move operator for tackling broadly defined Steiner tree problems (STPs). This operator complements the existing conventional move operators and could be adapted to solve many STP instances (corresponding to various Steiner tree problems), where (1) the optimal solution must be a minimum spanning tree, (2) each edge is associated with a uniform cost (this requirement can be relaxed to include instances with nearly uniform edge costs, using the techniques detailed in [28]), and (3) the input graph is of reasonable size (depending on the available memory of the used computer). We showed that with the aid of dynamic data structures, one can efficiently evaluate the feasibility of every possible swap-vertex move. Additionally, we designed an auxiliary evaluation function, which is able to discriminate different swap moves leading to the same move gain, in order to guide the search towards promising search regions. Our computational study carried out on three representative STP variants (i.e., the SPG, the PCSPG and the MWCS, all being part of the 11th DIMACS implementation challenge) demonstrated the usefulness of the swap-vertex operator and the auxiliary evaluation function for improved search performances. More generally, these techniques could be advantageously combined with other powerful search strategies to create effective algorithms for solving various Steiner tree problems.

## References

1. Cheng, X., Du, D.Z.: Steiner Trees in Industry. Springer, Berlin (2002)
2. Hakimi, S.L.: Steiner's problem in graphs. Networks **1**, 113–133 (1971)
3. Hwang, F.K.: On Steiner minimal trees with rectilinear distance. SIAM J. Appl. Math. **30**(1), 104–114 (1976)
4. Smith, W.D.: How to find Steiner minimal trees in Euclidean d-space. Algorithmica **7**(2/3), 137–177 (1992)
5. Johnson, D., Minkoff, M., Phillips, S.: The prize collecting Steiner tree problem: theory and practice. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 760–769 (2000)
6. Segev, A.: The node weighted Steiner tree problem. Networks **17**, 1–17 (1987)

7. Álvarez-Miranda, E., Ljubic, I., Mutzel, P.: The maximum weight connected subgraph problem. In: Jünger, M., Reinelt, G. (eds.) Facets of Combinatorial Optimization, pp. 245–270. Springer, Berlin (2013)

8. Ferreira, C.E., Filho, F.M.O.: Some formulations for the group Steiner tree problem. Electron. Notes Discr. Math. **18**(1), 127–132 (2004)

9. Salazar, J.J.: A note on the generalized Steiner tree polytope. Discr. Appl. Math. **100**(1–2), 137–144 (2000)

10. Chen, S.Y., Li, C.F., Chang, Y.W., Yang, C.L.: Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs. IEEE Trans. Comput. Aided Des. Integr. Circuit Syst. **27**(4), 643–653 (2008)

11. Charikar, M., Chekuri, C., Cheung, T., Dai, Z., Goel, A., Guha, S., Li, M.: Approximation algorithms for directed Steiner problems. J. Algorithms **33**(1), 73–91 (1999)

12. Imase, M., Waxman, B.M.: Dynamic Steiner tree problem. SIAM J. Discr. Math. **4**(3), 369–384 (1991)

13. Kurz, D., Mutzel, P., Zey, B.: Parameterized algorithms for stochastic Steiner tree problems. Math. Eng. Methods Comput. Sci. **7721**, 143–154 (2013)

14. Voß, S.: The Steiner tree problem with hop constraints. Ann. Oper. Res. **86**, 271–294 (1999)

15. Fu, Z.H., Hao, J.K.: Breakout local search for the Steiner tree problem with revenue, budget and hop constraints. Eur. J. Oper. Res. **232**(1), 209–220 (2014)

16. Fu, Z.H., Hao, J.K.: Dynamic programming driven memetic search for the Steiner tree problem with revenue, budget and hop constraints. INFORMS J. Comput. **27**(2), 221–237 (2015)

17. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations. Plenum Press, New York (1972)

18. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)

19. Shi, W., Su, C.: The rectilinear Steiner arborescence problem is NP-complete. SIAM J. Comput. **35**(3), 729–740 (2006)

20. Fischetti, M., Leitner, M., Ljubic, I., Luipersbeck, M., Monaci, M., Resch, M., Salvagnin, D., Sinnl, M.: 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

21. Gamrath, G., Koch, T., Maher, S. T., Rehfeldt, D., Shinano, Y.: 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 online at http://dimacs11.zib.de/workshop.html

22. Fonseca, R., Brazil, M., Winter, P., Zachariasen, M.: Faster exact algorithms for computing Steiner trees in higher dimensional Euclidean spaces. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4–5, (2014), available online at http://dimacs11.zib.de/workshop.html

23. El-Kebir, M., Klau, G. W.: 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

24. Minoux, M.: Efficient greedy heuristics for Steiner tree problems using reoptimization and supermodularity. INFOR **28**, 221–233 (1990)

25. Uchoa, E., Werneck, R.F.: Fast local search for Steiner trees in graphs. ACM J. Exp. Algorithms **17**(2), 2.2:1–2.2:22 (2012)

26. Pajor, T., Uchoa, E., Werneck, R.: A robust and scalable algorithm for the Steiner problem in graphs. Workshop of the 11th Dimacs Implementation Challenge, Providence, Rhode Island, December 4–5, (2014), available online at http://dimacs11.zib.de/workshop.html

27. Canuto, S.A., Resende, M.G.C., Ribeiro, C.C.: Local search with perturbations for the prize-collecting Steiner tree problem in graphs. Networks **38**, 50–58 (2001)

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

29. Burdakov, O., Kvarnstrom, J., Doherty, P.: Local search heuristics for hop-constrained directed 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

30. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco, California (2005)

31. Johnson, D. S, Koch, T., Werneck, R. F., Zachariasen, M.: 11th DIMACS Implementation Challenge in collaboration with ICERM: Steiner tree problems. http://dimacs11.zib.de (2013–2014)
32. Dittrich, M.T., Klau, G.W., Rosenwald, A., Dandekar, T., Müller, T.: Identifying functional modules in protein-protein interaction networks: an intergrated exact approach. Bioinformatics **24**(13), i223–31 (2008)
33. Rodriguez-Tello, E., Hao, J.K., Torres-Jimenez, J.: An improved simulated annealing algorithm for bandwidth minimization. Eur. J. Oper. Res. **185**(3), 1319–1335 (2008)
34. Rodriguez-Tello, E., Hao, J.K., Torres-Jimenez, J.: An effective two-stage simulated annealing algorithm for the minimum linear arrangement problem. Comput. Oper. Res. **35**(10), 3331–3346 (2008)