A New Crossover Mechanism for Genetic Algorithms for Steiner Tree Optimization

Qiongbing Zhang, Shengxiang Yang, *Senior Member, IEEE,* Min Liu, Jianxun Liu, and Lei Jiang,

***Abstract*—Genetic Algorithms (GAs) have been widely applied in Steiner tree optimization problems. However, as the core operation, existing crossover operators for tree-based GAs suffer from producing illegal offspring trees. Therefore, some global link information must be adopted to ensure the connectivity of the offspring, which incurs heavy computation. To address this problem, this paper proposes a new crossover mechanism, called Leaf Crossover, which generates legal offspring by just exchanging partial parent chromosomes, requiring neither the global network link information, encoding/decoding nor repair operations. Our simulation study indicates that GAs with leaf crossover outperform GAs with existing crossover mechanisms in terms of not only producing better solutions but also converging faster in networks of varying sizes.**

***Index Terms*—Tree optimization problem, genetic algorithm, crossover mechanism, leaf crossover.**

I. Introduction

STEINER tree optimization problems can be typically represented by edge-scheduled graphs with each edge annotated with a quantity measure specifying a specific requirement. Given a set of labeled objects as shown in Fig. I, the goal of a tree optimization problem is to find a subgraph that connects the start node and target nodes in this connected graph with the minimal summation of the quantity measures associated with edges. In Fig. 1, the start node is 1, the set of target nodes is $\{3,\ 5,\ 6,\ 9,\ 10\}$, and $\omega(i,j)$ associates some quantitative measure to the link between nodes i and j. The optimization requires an optimal interconnection between the start node and target nodes and the solution space is the set of trees spanning the start node and target nodes. Classic algorithms for tree optimization such as Prim's and Kruskal's, can not apply in many real scenarios, especially when they encounter Constrained Steiner Tree (CST) problems. In CST, the presence of two or more quantitative measures asks for a solution not only achieving the minimal summation of one index but also satisfying quantitative constraints on some other measures. As a famous intelligent computing tool, Genetic algorithms (GAs) [1], [2] have been widely used to solve various Steiner tree optimization problems [3]. Rojas and Mazar [4] presented a distributed GA to solve the Prize Collecting Steiner Tree Problem. Shi et al. [5] applied GAs for degree constrained minimum spanning trees in communication networks. Gen et al. [6] adopted GAs to design a supply chain network for production and distribution activities planning. Communication constitutes a large field, where GAs have found successful applications to solve associated Steiner tree optimization problems, including multicast traffic engineering in mobile ad hoc networks [7]–[9] and the Quality of Service (QoS) multicast problem [10], [11].

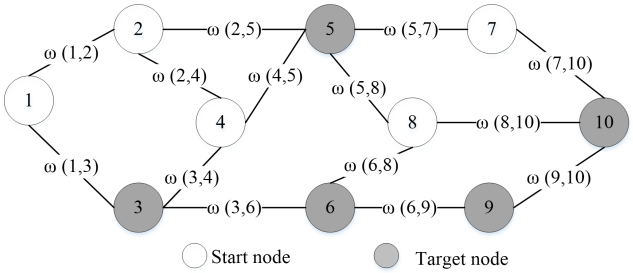


Fig. 1. A network topology.

Existing GAs for Steiner tree optimization problems can be grouped into the following two categories according to the representation of chromosomes:

1) GAs with direct tree encoding (GAs with DT) [10], [12], [13]. This class of GAs directly adopts trees as chromosomes.
2) GAs with indirect encoding. This class of GAs encodes trees into permutation strings as chromosomes. There are many coding schemes that have been proposed, such as the Prüfer encoding [14]–[16], Blob Code, Dandelion Code, Happy Code [17], MHappy Code [18] and Node-depth phylogenetic-based encoding [19].

In general, a GA first transfers trees to individuals, called chromosomes. Then, it iteratively generates new chromosomes by two operations, called crossover and mutation respectively, and evaluates the quality of chromosomes according to the fitness function. Since the probability of the occurrence of crossover is typically much larger than that of mutation, the crossover operator constitutes the core operation in GAs. Even though GAs for Steiner tree optimization have been widely applied in many fields, the crossover mechanism of GAs is still far from being satisfactory. As the core operation in GAs, the crossover operation generally only needs exchanging partial parent chromosomes to generate new chromosomes,

Manuscript received January 15, 2020; revised April 30, 2020; accepted June 18, 2020. This work was partially supported by the National Natural Science Foundation of China (NSFC) under Grant 61673331, and Shenzhen Science and Technology Program under Grant KQTD2016112514355531 (*Corresponding author: Shengxiang Yang*).

Qiongbing Zhang, Min Liu, Jianxun Liu, and Lei Jiang are with the School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan 411201, China (e-mail: mrtly2@whu.edu.cn).

Shengxiang. Yang is with the School of Computer Science and Informatics, De Montfort University, Leicester LE1 9BH, U.K. and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: syang@dmu.ac.uk).

e.g., in GAs for path optimization, combinational optimization and function optimisation. However, it is not the case for GAs applied to Steiner tree optimization problems, where when exchanging partial parent chromosomes to form new chromosomes, it is difficult to maintain/repair the legality of new trees. Loops and missing target nodes are highly probable to occur. Hence, existing crossover mechanisms adopt complex schemes to maintain the legality of offspring trees.

For GAs with DT, the crossover mechanism adopts shortest path algorithms such as Prim [20], [21], Dijkstra [22], to ensure the connectivity of offspring trees. These greedy algorithms need global network link information and involve complicated operations. GAs with indirect encoding facilitate crossover operations through a pretreatment of transferring parent trees into strings instead. This type of GAs adopts conventional string crossover mechanisms. But, it may generate illegal trees after decoding offspring chromosomes and require additional functions of detection and repair, diminishing the effectiveness of the algorithm. Furthermore, the repair function also adopts shortest path algorithms.

To address this problem, we propose a new crossover mechanism, called Leaf Crossover (LC), for GAs for tree problems. LC does not require an encoding scheme to transfer trees into strings, but takes trees as chromosomes directly. Unlike existing crossover mechanisms, LC generates offspring trees through only exchanging partial parent chromosomes. We take leaf nodes in parent trees as cut-in points to decompose parent trees into several paths. Then, the proposed crossover mechanism iteratively compares two corresponding paths which have the same leaf node. The better one will be used to form the offspring tree. We prove that LC can always generate legal offspring trees and requires neither repair functions nor greedy algorithms, which represents wide adaptability and less computations.

The rest of this paper is organized as follows. Section II presents the formulation of Steiner tree optimization problems. Section III provides a review of crossover mechanisms of tree-based GAs. The proposed crossover mechanism is introduced in Section IV. Section V presents a theoretical analysis for our proposed crossover mechanism. Section VI presents a detailed implementation of the proposed GA. Section VII experimentally evaluates various aspects of our algorithm. Some conclusions are given in section VIII.

II. Problem Description and Formulation

We denote a topology graph $G = \{V, E\}$ as a network containing a set of nodes V and a set of links E. The term $link(u, v) \in \mathbf{E}$ denotes the direct link connecting nodes u and v, and a positive weight $\omega(u, v)$ represents a scale value of physical sense such as the distance or cost of $link(u, v)$.

Let the source node be node s and the set of target nodes be $Targets$, where $\{s\} \cup Targets \subseteq V$. We also let $\Pi = \{T_1, T_2, \cdots, T_j, T_{j+1}, \cdots, T_n\}$ represent the solution space of all trees spanning s and $Targets$, where $T_j = \{V_j, E_j\} \subseteq G$, $V_j \subseteq V$ and $E_j \subseteq E$. For $\forall T_j \in \mathbf{\Pi}$, let

$$\omega(T_j) = \sum_{link(u,v)\in \mathbf{E_j}} \omega(u, v). \tag{1}$$

Then the well-known tree optimization problem can be formulated as follows.

1) If there is no constraint annotated to edges, we have

$$Min\{\omega(T) : T \in \mathbf{\Pi}\} \tag{2}$$

2) If there are constraints annotated to edges, $C_i(u, v), i = 1, \cdots, k, k \geq 1$; denotes a quantitative index of the constraints. For $\forall T_j \in \mathbf{\Pi}$, let

$$C_i(T_j) = \sum_{link(u,v)\in \mathbf{E_j}} C_i(u, v), i = 1, \cdots, k. \tag{3}$$

The optimization problem is:

$$Min\{\omega(T) : T \in \mathbf{\Pi}\}, \tag{4}$$

subject to:

$$C_i(T) \leq c_{i_0}, i = 1, \cdots, k; \tag{5}$$

where c_{i_0} is the threshold value of constraint C_i.

III. Review of GAs for Steiner Tree Optimization Problems

This section reviews crossover mechanisms of popular tree-based GAs. Crossover mechanisms for tree optimization have been widely applied in empirical modeling, decision tree, polynomial symbolic regression, data classification, syntax tree, etc. Dabhi and Chaudhary [23] surveyed and classified genetic programming (GP) with efficient crossover operators applied on empirical modeling. Crossover mechanisms avoiding illegal tree in this field include similar depth dependent crossover [24] and height-fair crossover. Šprogar proposed a novel homologous crossover operator for decision tree optimization [25]. Jabeen *et al.* [26] present a depth limited crossover in GP for classifier evolution. However, these problems can not be abstracted as the Steiner tree problem, which aims to find a subgraph that contains some specific nodes in edge-scheduled graphs.

Illegal trees in Steiner tree problems are mainly due to non-existent connection, loop and missing target nodes, while inefficient trees in GP for above problems are typically due to code bloat such as trees increase in complexity. Hence, the corresponding crossover mechanisms to avoid illegal trees in them are totally different. Diveev [27] proposed an attractive method to avoid illegal solutions in combinatorial optimization problem. It iteratively searches the small variation of a basic solution in a limited range to find a better solution. However, in order to obtain a solution code, it is necessary to perform additional calculations to determine variations to the basic solution.The additional calculations would be very large when applying it to a Steiner tree given that determining variations to a tree is more complicated than that to a single number sequence.

In Steiner tree optimization, for GAs with indirect encoding, there are two kinds of GAs: GAs with Prüfer and GAs with the sequence and topology encoding (GAs with ST) [11], [28]–[31], where the former is the basic and most famous encoding scheme and the latter is a kind of "Dandelion-like" [32] encoding scheme and very popular in the field of

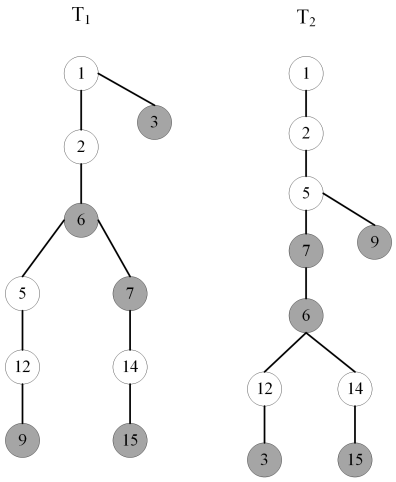


Fig. 2. Two Steiner trees.

communication multicasting. In fact, to our best knowledge, crossover mechanisms of all GAs with encoding are very analogous and have no significant difference. We outline crossover mechanisms as follows.

A. Crossover for GAs with DT

In this classic scheme, a chromosome is directly represented as a tree. During the operation of crossover, two randomly chosen parent chromosomes copy the same links and target nodes to the offspring directly. These nodes and links usually are not a connected tree but a series of separating subtrees. The next procedure is to connect these disconnected subtrees by iteratively connecting two randomly selected subtrees with a greedy algorithm until all subtrees are connected. Given two Steiner trees T_1 and T_2 as shown in Fig. 2, $s = 1$ and $Targets = \{3, 6, 7, 9, 15\}$, the crossover of GAs with DT proceeds as follows:

Step 1 : Obtain the subtree $T_{12} = \{\{s\} \cup Targets \cup V(E_1 \cap E_2), E_1 \cap E_2\}$, where $V(A)$ denotes the nodes contained in A for any $A \subset E$.

Step 2 : For any two separating trees $t_1 \in \mathbf{T}_{12}$ and $t_2 \in \mathbf{T}_{12}$, connect t_1 and t_2 into a tree by a shortest path algorithm such as Prim.

Step 3 : Repeat Step 2 until T_{12} is a connected tree.

B. Crossover for GAs with Prüfer

According to Cayley's theorem, Prüfer established a one-to-one correspondence between trees and node permutations [16]. For a tree, Prüfer presented an encoding scheme to convert it into two permutation strings: P_N and P_L, where P_N is the string of Prüfer number and P_L is the string of leaf nodes. The detailed encoding/decoding procedures are shown in [16], [33].

Standard string crossover mechanisms can be applied due to the representation of trees with strings. The most popular string crossover operator for GAs with Prüfer is the two-point crossover, where parent strings exchange gene fragments between two randomly selected positions. The offspring trees are obtained by decoding the generated new Prüfer strings.

However, randomly exchanging gene fragments from parents could make the resulting nodes connected to any node after decoding. Offspring trees are likely to be illegal, given that it is impossible for each node in the graph to have a direct link with every other node. To overcome this disadvantage, a detection procedure with some global link information is required to check the legality of these links and a repair function adopting the shortest path algorithm is needed if they are illegal. Therefore, this category of GAs needs a detection procedure and a repair function with greedy algorithms (such as Prim) to ensure the legality of offspring trees. In fact, all GAs with encoding for Steiner tree optimizations need the detection procedure and a repair function.

C. Crossover for GAs with ST

The Prüfer scheme is not robust since changing even an element of a Prüfer number could change the tree significantly [7]. Many encoding schemes have been proposed to address this disadvantage. However, crossover mechanisms of these GAs also need global link information and heavy computation to ensure the legality of offspring trees. We take GAs with ST [11] as an instance, which are very popular in the field of communication multicasting. In this scheme, a tree has to be transferred to a position indexed tree before the ST encoding.

The ST encoding scheme converts a tree into two permutation strings: the string s is a sequence of node numbers and the string t is used to encode the topology of the tree. Like the Prüfer scheme, the conventional string crossover mechanisms can then be applied to two strings, e.g., the two-point crossover. However, this category of GAs also needs a detection procedure and a repair function as it may generate illegal trees due to the same reason as GAs with Prüfer.

IV. Proposed new crossover mechanism

A greedy algorithm is usually required to combine the involved separating subtrees during the crossover process for GAs with DT, which needs to compare all neighboring nodes of a subtree to derive the next node for connecting two subtrees. Therefore, GAs with DT need the global network link information and heavy computations. A first glance implies that GAs with indirect encoding would simplify the crossover operation since it transforms the tree into two strings so that standard string crossover operators can be applied. However, a closer look shows that this is indeed not the case. On the one hand, they need massive computations for iterative encoding/decoding and for detecting the legality of each chromosome. On the other hand, it also needs global network link information and greedy algorithms to repair offspring trees when necessary.

We want to design a new convenient crossover mechanism that only require to exchange partial parent chromosomes to generate new legal chromosomes without a detection procedure or other algorithms, just like GAs in other fields. For this purpose, we propose a novel crossover mechanism, called Leaf Crossover (LC), which has two main procedures.

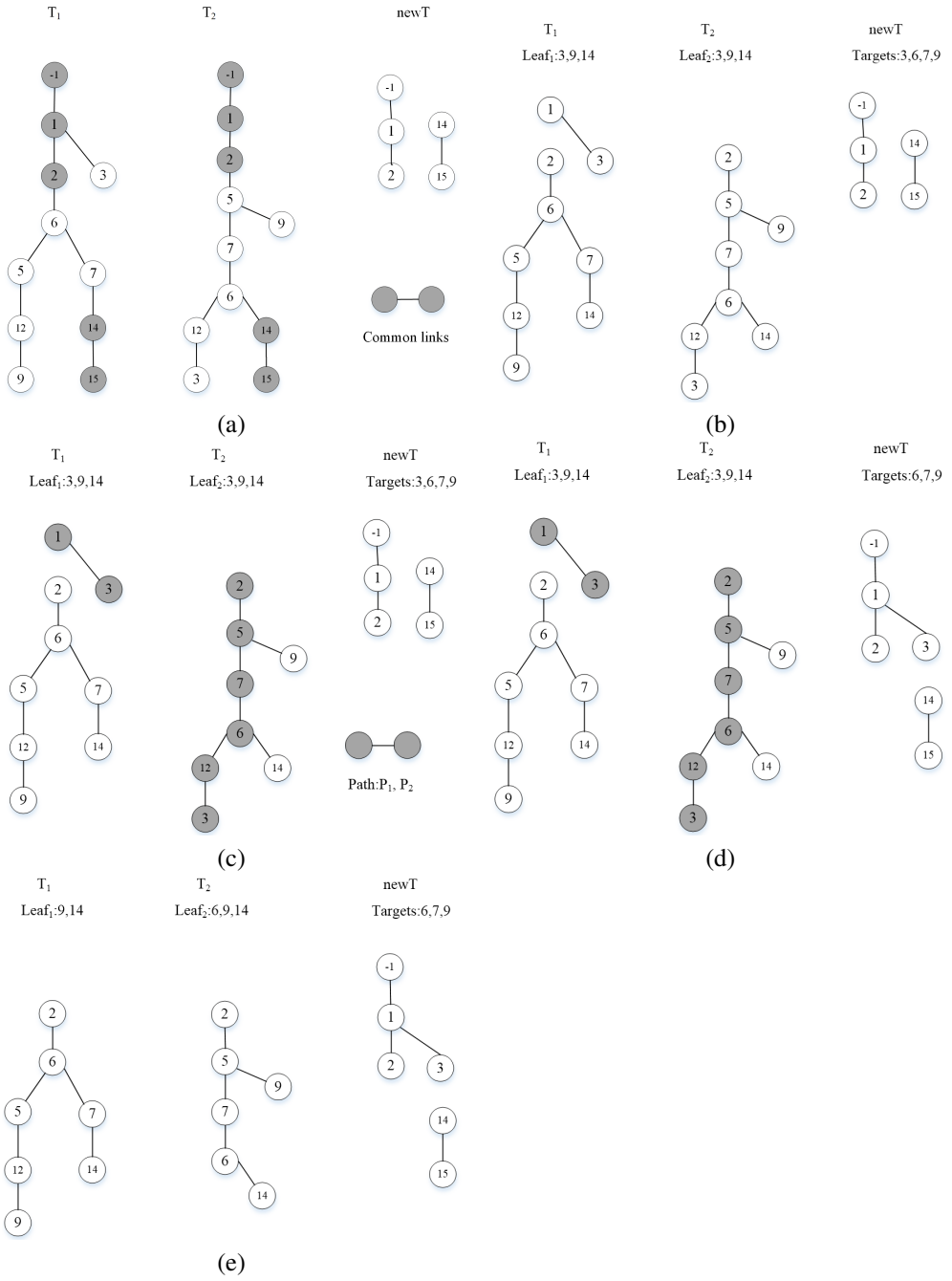


Fig. 3. Illustration of LC. (a) Copy same links to the child (Procedure I). (b) Select a same leaf node (Procedure II). (c) Generate two paths (Procedure II). (d) Add the better path into the child (Procedure II). (e) Remove redundant links (Procedure II).

In the first procedure, LC adds a link $(s, -1)$ for the root node s and then removes the same links and copies them to the child directly: $newT = \{V(E_1 \cap E_2), E_1 \cap E_2\}$. The tip of adding a link $(s, -1)$ for the root node s is to ensure that $newT$ cannot be empty. This procedure is analogous to that of GAs with DT. It has two advantages: 1) it avoids encoding/decoding operations and 2) it saves operations by passing same links between parent trees to the offspring tree directly. Especially, when we perform crossover on two same trees, it can get the same offspring tree directly to avoid the next procedure. Fig. 3(a) shows an example of the first procedure of LC, the same links between T_1 and T_2 are $\{(1, -1), (2, 1), (15, 14)\}$. Therefore, $newT = \{\{15, 14, 2, 1, -1\}, \{(1, -1), (2, 1), (15, 14)\}\}$.

After the first procedure, both parent trees and the offspring tree are sets of separating subtrees. It seems that there are many residual links, which are disordered. It is quite challenging to select the essential and high-quality links from them to constitute a legal offspring tree. GAs with DT abandons residual links entirely and adopts shortest path algorithms to connect the subtrees of the offspring tree. Fortunately, we find a scheme of taking leaf nodes as cut-in points to select the essential and high-quality links from residual links. As Fig. 3(b) shows, after the first procedure, leaf nodes of T_1 and

T_2 are $leaf_1 = \{3, 9, 14\}$ and $leaf_2 = \{3, 9, 14\}$ respectively. Meanwhile, since $newT$ has the target node 15, the list of target nodes is $Targets = \{3, 6, 7, 9\}$.

In LC, the second procedure has two parts: one is to add the high-quality path into the offspring tree and the other is to delete redundant links in parent trees. To add the high-quality path into $newT$, we first randomly choose a node v from $leaf_1 \cap leaf_2$ and find the paths P_1 and P_2 from v back to the root node in T_1 and T_2 respectively. For instance, as Fig. 3(c) shows, if $v = 3$, $P_1 = \{(3, 1)\}$, $P_2 = \{(3, 12), (12, 6), (6, 7), (7, 5), (5, 2)\}$. The target of this part operations is to determine a path which would be added into the offspring tree. Hence, we should find the junction point μ connecting the path to $newT$. Let P denote a path and V_P represent the nodes in P. We also define $Child(E_T) = \{v|$the child node of $link(v, u), link(v, u) \in \mathbf{E}_T\}$. The junction point μ should satisfy the following condition:

$$\mu \in \mathbf{C}hild(E_{newT}) \cap V_p \tag{6}$$

For instance, in Fig. 3(c), $Child(E_T) = \{1, 2, 15\}$. Hence two junction points in P_1 and P_2 are $\mu_1 = 1$ and $\mu_2 = 2$ respectively. Noting that the small tip of adding a link $(s, -1)$ for the root node s in the first procedure makes s satisfy the condition $s \in \mathbf{C}hild(E_{newT})$. Meanwhile, if there is more than one node in $Child(E_{newT}) \cap V_p$, LC just selects the one that is the nearest to the leaf node as the junction point and cuts the path to only retain links between the leaf node to the junction point.

For convenience, we have the following notations to define this connective path and the cut operation.

- $P(v, u)$: The path connecting v to u.
- $depth(T, v)$: The depth of v in tree T, where $v \in \mathbf{V}(T)$.
- $Cut(v, u, P) = P(v, u)$: For any path P, $Cut(v, u, P)$ denotes the path by removing all edges of P expect for edges connecting v to u, where $v \in \mathbf{V}_p$ and $u \in \mathbf{V}_p$.
- $Connect(T_1, newT, v) = \{P(v, \mu)|P(v, \mu) \subseteq T_1,\ \mu \in \mathbf{C}hild(E_{newT}),\ depth(T_1, \mu) < depth(T_1, v)$ and $\{V_p - \mu\} \cap Child(E_{newT}) = \emptyset\}$.

Here, $Connect(T_1, newT, v)$ denotes the path $P(v, \mu)$ in tree T_1, satisfying the condition that μ is the only node in $Child(E_{newT}) \cap V_p$. For instance, $Connect(T_1, newT, 9) = P(9, 2) = \{(9, 12), (12, 5), (5, 6), (6, 2)\}$, where T_1 and $newT$ are shown in Fig. 3(b). For the notation $Cut(v, u, P)$, we have $Cut(9, 6, P) = Path(9, 6) = \{(9, 12), (12, 5), (5, 6)\}$ with a given path $P = \{(9, 12), (12, 5), (5, 6), (6, 2)\}$.

We have two connective paths P_1 and P_2 now. The last step of this part of operations is comparing the qualities of P_1 and P_2 and adding the better one into $newT$. According to Eq. (2), the quality function is listed as follows.

$$F'(T) = \omega_0 - \omega(T), \tag{7}$$

where ω_0 is a sufficiently large constant to make $F'(T)$ positive (Paths also can be taken as trees).

Fig. 3(d) shows the condition when $F'(P_1) > F'(P_2)$, we add P_1 into $newT$ and delete node 3 from $Targets$ since it is contained in P_1. Since P_1 and P_2 were compared and the better one has been added to the offspring tree, the links of these two paths may be redundant and should be removed from parent trees. All links of P_1 have been added to $newT$, we just remove P_1 from T_1. For P_2, the removing procedure should be careful in order not to affect other leaf nodes connecting to the offspring tree. For instance, if we have removed any link of (6,7), (7,5) and (5,2), there is no path connecting leaf node 14 to $newT$. Meanwhile, removing nodes 6 and 7 may lead to the offspring tree missing target nodes because they are contained in the list of targets. The rules for removing redundant links are summarized as follows.

Removing Rules. For a path $P \subset T$, the rules for removing P from the parent tree T are listed as follows.

1) If P is chosen to be added into the new tree, then E_P which denotes all edges in P is removed from T. Furthermore, if node $v \in \mathbf{T}argets \cap Child(E_p)$, set $Targets = Targets - v$;
2) If P is not added into the new tree, remove it from T as follows.

   Step 1 : $v = Leaf(P)$, where $Leaf(P)$ denotes leaf nodes in P;
   Step 2 : $T = T - v$ and $P = P - v$;
   Step 3 : $v = Leaf(P)$. If $v \in \mathbf{T}argets$ or $|child(v)| > 1$ (where $|child(v)|$ means the number of child nodes of v), stop the iteration; otherwise, go to Step 2. Furthermore, if $v \in \mathbf{T}argets$, set $leaf_T = leaf_T + v$.

Fig. 3(e) shows the procedure of removing P_1 and P_2 from T_1 and T_2. T_1 removes all edges of P_1 while T_2 removes nodes in P_2 from the leaf nodes to top by the iteration procedure. The iterations are stopped on node 6 since $|child(6)| > 1$. Meanwhile, as $node\ 6 \in \mathbf{T}argets$, it follows that $leaf_2 = \{leaf_2 + 6\} = \{6, 9, 14\}$.

Then, one iteration is over and next operations of LC just iteratively select a same leaf node $v \in \mathbf{l}eaf_1 \cap leaf_2$ to repeat the above operations. The detailed description which is well illustrated with many easy to parse figures is shown in the appendix. The complete LC procedure on parent trees T_1 and T_2 is outlined in the following steps.

Step 1 : Add the link $(s, -1)$ to T_1 and T_2. Obtain the generated new tree: $newT = \{V(E_1 \cap E_2), E_1 \cap E_2\}$, and update $T_1 = T_1 - newT$ and $T_2 = T_2 - newT$.
Step 2 : Maintain two lists $leaf_1 = Leaf(T_1)$ and $leaf_2 = Leaf(T_2)$, and randomly select a node $v \in \mathbf{l}eaf_1 \cap leaf_2$.
Step 3 : $P_1 = Connect(T_1, newT, v)$ and $P_2 = Connect(T_2, newT, v)$.
Step 4 : If $P_1 \cap P_2 - v \neq \emptyset$, select node $u = min\{Hop(v, u, T_1) + Hop(v, u, T_2)|u \in \{P_1 \cap P_2 - v\}\}$, then $P_1 = Cut(v, u, P_1)$, $P_2 = Cut(v, u, P_2)$, $leaf_1 = leaf_1 + u$, and $leaf_2 = leaf_2 + u$.
Step 5 : For P_1, check if there is a loop in $newT \cup P_1$. If so, then $t_1' = \phi$ and $F'(t_1') = 0$. Do the same for P_2.
Step 6 : If $P_1 \cup P_2 = \phi$, the crossover is over and no offspring tree is obtained; otherwise, compare $F'(P_1)$ with $F'(P_2)$. If $F'(P_1) > F'(P_2)$, then $newT = \{newT + P_1\}$; otherwise, $newT = \{newT + P_2\}$.

Remove P_1 and P_2 from T_1 and T_2 respectively according to *Remove Rules*.

Step 7 : If $leaf_1 \cap leaf_2 \neq \phi$, go to Step 2.

Step 8 : If $T_1 = \phi$ or $T_2 = \phi$, go to Step 10.

Step 9 : Check if there is a loop in $newT \cup T_1$: if so, then $T_1 = \phi$ and $F'(T_1) = 0$. Do the same for T_2. If $T_1 \cup T_2 = \phi$, the crossover is over and no offspring tree is obtained; otherwise, compare $F'(T_1)$ with $F'(T_2)$. If $F'(T_1) > F'(T_2)$, $newT = \{newT + T_1\}$; otherwise, $newT = \{newT + T_2\}$.

Step 10: Return the new tree $newT$.

V. Validity Analysis

The above overall description of the LC operator shows that every iteration in crossover begins from the leaf node, which is why we term the proposed crossover mechanism as the Leaf Crossover. This section aims to present a theoretical analysis for four kinds of crossover mechanisms to show why LC can be more effective than others. Here, we validate the effectiveness from two aspects: legal offspring trees and time complexity.

A. Legal offspring trees

The demonstration that LC can always generate legal offspring trees is given as follows. A legal Steiner tree should satisfy the following conditions.

1) It contains all target nodes as well as the root node.
2) No loop is present in the tree.
3) It is a connected tree.

Below we show that a tree generated by LC satisfies the above conditions.

1) For the root node s, we add the edge $(s, -1)$ in each tree so that there always exists a root node in the first step: $newT = \{V(E_1 \cap E_2), E_1 \cap E_2\}$. Meanwhile, the $Remove\ Rules$, which does not remove the node in $Targets$ while not contained in $newT$, ensures that the generated new tree should contain all target nodes when either $T_1 = \phi$ or $T_2 = \phi$.
2) There is no loop in $newT$ due to the check module in Step 5.
3) Assuming that there exists a separating subtree t' in the final $newT$ and the root of t' is u, there are 3 conditions that may generate a subtree in LC: Steps 1, 3 and 4. If t' is generated by Step 1: $newT = \{V(E_1 \cap E_2), E_1 \cap E_2\}$, node u will be a node in two leaf lists which would be selected to connect another subtree as the leaf node in the subsequent iterations. If t' is generated by the $Connect()$ function in Step 3, u should be connected to $newT$ by definition. The last condition is that t' is generated by the $Cut()$ function in Step 4. In this condition, u will also be added into two leaf lists which would be selected to connect another subtree as the leaf node in the subsequent iterations. Therefore, $newT$ will be a connected tree when either $T_1 = \phi$ or $T_2 = \phi$.

B. Time complexity

Here, we compare the time complexities of different crossover mechanisms. Consider a graph $G = (V, E)$ as a network containing a set of nodes V and a set of links E, $|V| = n$ and $|E| = m$. Let Π be the set of feasible candidate solution trees, T_1 and $T_2 \in \mathbf{\Pi}$ with the number of nodes being L_1 and L_2 respectively. The time complexity analysis of different crossover mechanisms on T_1 and T_2 is outlined here.

1) GAs with DT: The crossover operation of GAs with DT has two steps. The first step is to obtain the subtree set: $\{E_1 \cap E_2\}$. Since each link in one tree should be checked whether it also belongs to another tree, the time complexity of $\{E_1 \cap E_2\}$ can be easily calculated as follows:

$$O(\frac{(L_1 - 1)(L_2 - 1)}{2}),$$

where $L_1 - 1 = |E_1|$ and $L_2 - 1 = |E_2|$. The second step is to connect subtrees by the shortest path algorithms. The most widely used algorithms are Prim and Dijkstra, which have the same time complexity $O(Prim)$. $O(Prim)$ is related to the data structure, attaining the minimal value $O(m+nlogn)$ [30] and the maximal value $O(n^2)$ [29]. So, the time complexity of crossover for GAs with DT is:

$$\begin{aligned} &O(\frac{(L_1 - 1)(L_2 - 1)}{2}) + O(k(Prim)) \\ &\approx O(\frac{L_1 L_2}{2}) + O(k(Prim)); \end{aligned} \tag{8}$$

where $k + 1$ denotes the number of separating trees.

2) GAs with Prüfer: The crossover operation for GAs with Prüfer includes encoding, two-point crossover, decoding, detection and repair. The encoding scheme transfers all links to the digits of the Prüfer string and the time complexity is:

$$O(L_1 + L_2 - 2),$$

where $L_1 + L_2 - 2 = |E_1| + |E_2|$. The two-point crossover aims to find the gene fragments and insert them into each other. Two strings P_{N1} and P_{N2} need a traversal to find two random positions to exchange gene fragments. The time complexity is:

$$O(L_1 - 2 + L_2 - 2),$$

where $L_1 - 2$ and $L_2 - 2$ are the lengths of Prüfer string of T_1 and T_2 respectively. The decoding procedure traverses Prüfer strings to generate offspring trees. The time complexity is also:

$$O(L_1 - 2 + L_2 - 2),$$

The detection procedure should detect every link in the offspring tree to check its legality. The time complexity is:

$$O(L_0 - 1),$$

where L_0 denotes the number of nodes in the offspring tree and $L_0 - 1$ is the number of links in the offspring tree. The last operation is to adopt a shortest path algorithm to repair the illegal offspring tree. Like GAs with DT, the time complexity is:

$$O(k'(Prim));$$

where k' denotes the number of illegal links. Therefore, the time complexity of this kind of crossover is:

$$\begin{aligned} &O(3L_1 + 3L_2 + L_0 - 11) + O(k(Prim)) \\ &\approx O(3L_1 + 3L_2 + L_0) + O(k(Prim)); \end{aligned} \tag{9}$$

3) GAs with ST: The ST encoding transfers each parent tree into two strings and the time complexity is:

$$O(L_1 + L_2).$$

The crossover mechanism of ST is also conventional string crossover mechanism. Noting that GAs with ST do two-point crossover on both s and t and the length of ST string is $2L$, where $L = |V_T|$. Analyzing in the same way, the total time complexity of the crossover of GAs with ST is:

$$\begin{aligned} &O(5L_1 + 5L_2 + L_0 - 1) + O(k''(Prim)) \\ &\approx O(5L_1 + 5L_2 + L_0) + O(k''(Prim)); \end{aligned} \tag{10}$$

where k'' denotes the number of illegal links. Compared with GAs with Prüfer, GAs with ST overcome a poor property that a small change in a string generally changes many edges in the tree it represents. Hence, generally $k''<k'$ and the time complexity of the crossover of GAs with ST is lower than that of GAs with Prüfer.

4) GAs with LC: The first step on LC is analogous to that of GAs with DT. So, the time complexity of the first step is:

$$O(\frac{(L_1 - 1)(L_2 - 1)}{2}).$$

The next operations are the iterations of Steps 2-6. We assume that the crossover terminates after p iterations. The main operations in the iteration contain $Connect()$, checking for $\{P_1 \cap P_2 - v\}$, checking loops for $newT \cup P$ and the removing procedure. The first operation in the iteration is the $Connect()$ function, which needs a traversal from the leaf node to the top to find a node contained in the offspring tree. In Fig. 3, T_1 and T_2 should find a path connecting the leaf node to $newT$, and the time complexity is:

$$O(\frac{L'_0}{2}\frac{d_1}{2}) + O(\frac{L'_0}{2}\frac{d_2}{2}),$$

where L'_0 is the number of the nodes in the $newT$ at this moment and d_1 and d_2 are the degree of T_1 and T_2 respectively. After p iterations, the total time complexity is:

$$O(p\frac{L'_0}{2}\frac{d_1}{2}) + O(p\frac{L'_0}{2}\frac{d_2}{2})<O(\frac{L_0}{2}\frac{d_1}{2}) + O(\frac{L_0}{2}\frac{d_2}{2}).$$

Then, LC needs to check if $\{P_1 \cap P_2 - v\} \neq \phi$. The time complexity is:

$$O(\frac{(L'_1)(L'_2)}{2}),$$

where L'_1 and L'_2 are the number of nodes in V_{p_1} and V_{p_2} at this moment. After p iterations, the total time complexity is:

$$O(p\frac{(L'_1)(L'_2)}{2})<O(\frac{(L_1)(L_2)}{2}).$$

Next, the mechanism checks if a loop exists in $newT \cup P_1$ and $newT \cup P_2$. This check procedure for $newT \cup P_1$ needs a traversal from the leaf node of P_1 to the top of $newT$ to find if any loop exists. For instance, in Fig. 3(d), $P_2 = \{(3, 12), (12, 6), (6, 7), (7, 5), (5, 2)\}$, the check module traverses only the path {(3, 12), (12, 6), (6, 7), (7, 5), (5, 2), (1, -1)} to find if any loop exists. The time complexity of the check procedure is:

$$O(L'_1 + \frac{d_0}{2}) + O(L'_2 + \frac{d_0}{2}).$$

Similarly, the time complexity after p iterations is:

$$O(pL'_1 + p\frac{d_0}{2}) + O(pL'_2 + p\frac{d_0}{2})<O(L_1 + L_0) + O(L_2 + L_0)$$

The final procedure is the removing procedure. Two parent trees remove almost all links when crossover is over. Hence, the time complexity of p iterations is:

$$O(L_1 + L_2).$$

In summary, the total time complexity of LC is less than

$$\begin{aligned} &O(L_1L_2 + \frac{L_0(d_1 + d_2)}{4} + 2L_1 + 2L_2 + 2L_0) \\ &\approx O(L_1 + d_0)(L_2 + d_0). \end{aligned} \tag{11}$$

Hence, the time complexity of LC is positively correlated not with the graph size but with the length of parent trees. Meanwhile, in general cases, Eq. (11) is much less than $kO(Prim)$.

To summarize, as compared with the three popular crossover mechanisms, our crossover mechanism needs less computations with added benefits of not requiring a shortest path algorithm, a detection procedure, a repair function, global link information, and encoding/decoding operations.

VI. Implementation of GA with Leaf Crossover

The main design of a GA includes population initialization, fitness function, selection, crossover and mutation. The population in Steiner tree optimization models is mostly initialized with the randomized depth-first search algorithms [34]. Like GAs with DT, the proposed GA takes the tree as the chromosome directly to avoid encoding/decoding operators. Therefore, we use randomized depth-first search algorithms to generate a set of Steiner trees in the population initialization stage.

A. Fitness function

The fitness function has two different equations according to whether there are constraints annotated to edges.

1) If there is no constraint annotated to edges, we just take the function $F'(T)$ in Section 4 as the fitness function.
2) If there are constraints annotated to edges, a common approach of constrained Steiner tree optimization problems is to use the penalty technique in fitness function design [10]. We use a simple function as the fitness function of the tree T:

$$F(T) = \delta(T)F'(T) = \delta(T)(\omega_0 - \omega(T)),$$

where

$$\delta(T) = \begin{cases} \alpha, if\ T\ does\ not\ satisfy\ constraints \\ 1, if\ T\ satisfies\ constraints \end{cases},$$

where α is the penalty factor usually restricted to the interval (0, 0.5].

B. Selection

Currently, there are two widely used types of selection schemes: proportionate and ordinal-based selection [35]. In this study, we implement both type mechanisms: roulette wheel (RW) and tournament pick (TP). In this paper, the size of the tournament pool is 5 and the probability P_i with which a tree T_i is selected in the roulette wheel is given as follows:

$$P_i = \frac{F(T_i)}{\sum_{j=1}^{popsize} F(T_j)}, \tag{12}$$

where $popsize$ means the population size.

C. Crossover and mutation

The proposed crossover is defined in Section 4. Here, we present the most popular mutation strategy. Oh et al. [36] proposed a new mutation scheme for tree-based GAs. According to the mutation probability P_m, this mutation strategy randomly selects a subset of nodes and breaks the multicast tree into some separating subtrees by removing all links incident to the selected nodes. Then, these separating subtrees are connected into a new multicast tree by randomly selecting a least-weight path between them.

VII. Experimental Study

We have conducted experiments to compare the performance of GAs with the proposed crossover operator with the existing popular GAs introduced in Section 3. We compare GAs with the Prüfer, DT, ST, and LC on 9 benchmarks (G_1-G_9) taken from the OR library [26]: steinb2 ($G1$), steinb3 ($G2$), steinb5 ($G3$), steinb13 ($G4$), steinb16 ($G5$), steinb18 ($G6$), steinc8 ($G7$), steinc13 ($G8$), steind8 ($G9$), where the number of nodes, links and target nodes are: (50, 63, 13), (50, 63, 25), (50, 100, 13), (100, 125, 17), (100, 200, 17), (100, 200, 50), (500, 1000, 83), (500, 2500, 83), and (1000, 2000, 167) respectively. All GAs are implemented in C language and the execution environment is: CPU (Intel(R) Core(TM) i7-4790); RAM (8 G); Operating system (Windows 7). For a better comparison, we use identical GA operators except the crossover operators. The algorithms are terminated either when the population does not change during 10 consecutive generations or the iteration number reaches 200.

Three sets of experiments were carried out. The first set of experiments use G_1-G_6 to verify the mathematical analysis in Section 5, which shows that the time complexity of LC is much less than that of other crossover mechanisms. Meanwhile, this set of experiments also show GAs with LC execute faster than GAs with other crossover mechanisms. The second set of experiments compares the performance of four kinds of GAs on G_1-G_6. The third set of experiments checks the performance of GAs on large scale network graphs G_7-G_9.

A. Experiments regarding the executing time

We first conduct experiments under typical parameter options, of which the crossover and mutation probabilities are set as 0.6 and 0.05 respectively, and the population size is set to 50 on G_1-G_3 and 100 on G_4-G_6. First, to verify the mathematical analysis shown in Section 5, we record the total time of generating 1000 offspring trees by each crossover operator on randomly generated trees from G_1-G_6. In the previous analysis, the time complexity of LC is much less than that of other crossover mechanisms. Moreover, the time complexity of LC is not positively correlated with the graph size but with the length of parent trees while other crossover mechanisms are positively correlated with the graph size. As shown in Table I, the executing time of LC is much less than that of other crossover operators and this advantage is more highlighted for increasingly complicated network graphs. Therefore, the experimental results are consistent with the theoretical analysis.

TABLE I
THE SUM TIME OF FOUR CROSSOVER OPERATORS DURING 1000 CROSSOVER OPERATIONS

	G_1	G_2	G_3	G_4	G_5	G_6
LC	**0.4009**	**0.4742**	**0.7208**	**0.7909**	**1.4977**	**1.8049**
DT	0.7711	0.9829	1.1419	1.3665	6.5536	6.8875
ST	1.0093	1.3743	1.2761	1.5413	5.5068	6.1840
Prüfer	1.5007	1.7916	1.5998	3.1403	10.2367	17.2616

TABLE II
THE AVERAGE EXECUTING TIME OF GAS WITH FOUR CROSSOVER OPERATORS ON GENERAL GRAPHS

	G_1	G_2	G_3	G_4	G_5	G_6
LC	**0.1236**	**0.1658**	**0.1629**	**0.5920**	**0.9561**	**2.5776**
DT	0.1551	0.7603	0.1998	1.2214	4.9260	3.7842
ST	0.7575	0.5190	1.2351	8.3627	16.5940	18.8310
Prüfer	0.9039	3.3267	1.8394	11.0539	31.6059	50.1690

TABLE III
THE FITNESS EVALUATION TIMES OF GAS WITH FOUR CROSSOVER OPERATORS ON GENERAL GRAPHS

	G_1	G_2	G_3	G_4	G_5	G_6
LC	1734	**1786**	2065	5145	**5711**	6519
DT	**1596**	3388	**1455**	**4470**	6695	**5266**
ST	4095	4479	3621	9901	12292	10623
Prüfer	6031	8630	6283	16178	18608	17914

Table II presents the executing time of GAs on each network instance. The result is consistent with Table I, which illustrates that GAs with LC are significantly faster than popular GAs. Table III counts the average number of fitness evaluations required by GAs for achieving convergence on G_1-G_6. From the table, we can see that the GA with Prüfer requires the most number of evaluations for convergence, the GA with ST requires the second large number of fitness evaluations, while both LC and DT versions of GAs converge roughly at the same speed with the minimal number of fitness evaluations. Noting that, unlike other GAs, the executing time of GAs with DT does not increase with increasingly complicated network graphs. This is because GAs with DT take the shortest path algorithm as the main operation in crossover. Hence, the executing time is much affected by the relative localities of target nodes. If target nodes are closer to each other, the shortest path algorithm is easy to get the next target node and

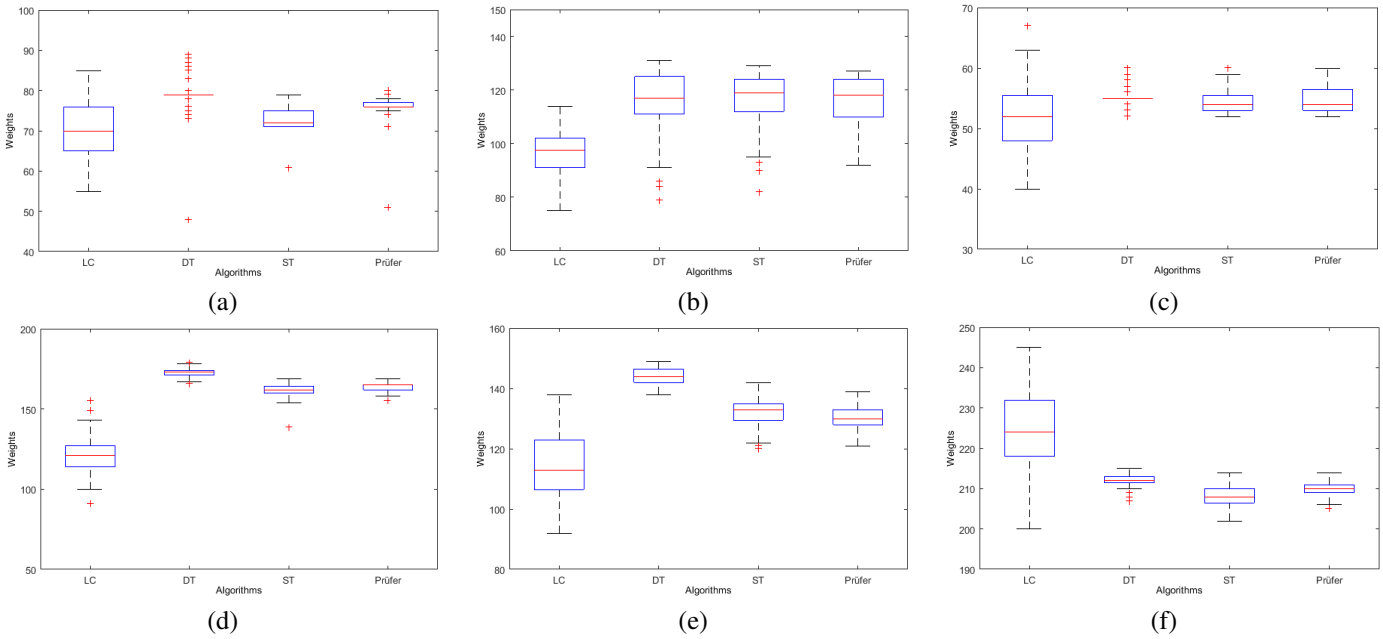


Fig. 4. Weights obtained by GAs under typical parameter options. (a) Results on G_1. (b) Results on G_2. (c) Results on G_3. (d) Results on G_4. (e) Results on G_5. (f) Results on G_6.

TABLE IV
EXPERIMENTAL RESULTS ON G_6 IN THE CASE OF GAS WITH THE POPULATION SIZE OF 200

	LC	DT	ST	Prüfer
Exec Time(s)	**5.7841**	9.3137	42.5007	83.8180
Mean weights	207.17	211.10	**207.11**	208.36
Min weights	**186**	202	201	201

Exec Time: the execution time; Min weights: the minimal weights

GAs with DT execute fast. When target nodes are far from each other, the shortest path algorithm would spend much time in getting the next target node and GAs execute slowly.

B. Performances on general graphs

This set of experiments compares the performance of four kind GAs. Fig. 4 records the weights obtained by GAs with typical parameter options on G_1-G_6. From the figure, we can see that the GA with LC performs best on G_1-G_5, it is only outperformed on G_6. This is because G_6 has the most complicated tree optimization problems (100,200,50), and the population size set as 100 is too small to adapt to this complicated graph. However other three GAs can to some degree offset this disadvantage with greedy algorithms. The repair function adopt global link information and greedy algorithms to handle non-existent links in offspring trees, which has the same effect as the mutation operators. This means the repair function has a side effect that is equivalent to enlarge the mutation probability of GAs. We then increase the population size to 200 and conduct a simulation for GAs on G_6. Table IV records the results of four GAs with a population size of 200 on G_6. From the table, we can see that LC and ST versions of GAs get almost the same mean weights while other two GAs obtain larger mean weights. Meanwhile, GAs with LC performs much better in terms of the execution time and minimal weights.

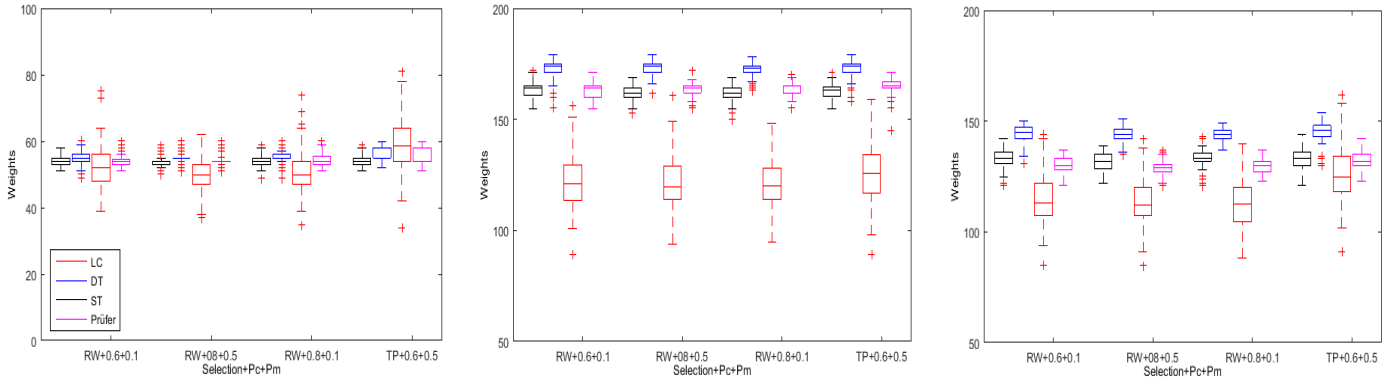
We use graphs G_3-G_5 to check the performance of GAs with different parameter combinations regarding the selection operator, crossover probability, and mutation probability. Four parameter combinations are checked: RW+0.6+0.1, RW+0.8+0.05, RW+0.8+0.1 and TP+0.6+0.05. The final weights obtained by GAs are presented in Fig. 5 and the execution time is recorded in Table V. From Fig. 5, we can see that GAs with LC outperform other GAs on almost all situations except the situation with TP+0.6+0.05 on G_3. Table V shows that GAs with LC requires much less execution time than others.

In summary, the experimental results for GAs with various parameter options show that GAs with LC have better performance than GAs with other three crossover mechanisms.

C. Performances on large scale graphs

Finally, we check the performance of GAs on large scale network network graphs G7-G9 respectively. The parameter combination is set as TP+0.6+0.05 and the population size is 500. Since the solution space and the population size are huge in large scale graphs, it is difficult for GAs to converge to a same solution. A common approach is to choose the best solution as the output solution.

Table VI presents the average execution time per generation in the first few generations. Noting that we did not execute GAs with Prüfer on large scale graphs since it needs too long execution time. We can see that only GAs with LC has an acceptable execution time, which is much less than that of other two GAs. GAs with LC need about 2.7s, 5.8s, and 32.6s to execute one generation on G_7, G_8, and G_9, respectively, while GAs with DT and GAs with ST need more than 550s to execute one generation on G_9. Since execution times of GAs



(a) (b) (c)

Fig. 5. Weights obtained by GAs under different parameter combinations. (a) Results on G_3. (b) Results on G_4. (c) Results on G_5.

TABLE V
THE EXECUTING TIME PER GENERATION OF GAS WITH DIFFERENT PARAMETER COMBINATIONS (S)

	RW+0.6+0.1				RW+0.8+0.05			
	DT	ST	Prüfer	LC	DT	ST	Prüfer	LC
G_3	0.2531	0.9786	1.5013	**0.2004**	0.2221	1.4910	2.2504	**0.1951**
G_4	1.4063	7.0636	9.8183	**0.6014**	1.4956	11.1406	14.690	**0.5932**
G_5	4.2033	11.4727	24.9927	**1.2297**	5.5985	19.6837	41.0905	**1.1070**
	RW+0.8+0.1				TP+0.6+0.05			
	DT	ST	Prüfer	LC	DT	ST	Prüfer	LC
G_3	0.2589	1.1561	1.9084	**0.2333**	0.2443	1.0717	1.6073	**0.1683**
G_4	1.5360	8.3250	12.0812	**0.6574**	1.3721	7.3503	7.1442	**0.5867**
G_5	5.3461	13.8622	30.7497	**1.1607**	3.2260	12.8683	18.9353	**1.0176**

TABLE VI
THE EXECUTING TIME PER GENERATION OF GAS ON LARGE SCALE GRAPHS (S)

	LC	DT	ST
G_7	**2.7466**	37.5887	61.2815
G_8	**5.8378**	135.7780	150.3803
G_9	**32.5727**	575.8421	586.1900

with DT and ST are too long in our execution environment to complete the statistical test, this set of experiments was done on the supercomputing system in the Supercomputing Center of Wuhan University. Fig. 6(a-c) records best weights obtained by GAs in the first 30 generations. According to this figure, GAs with DT own the quickest converging, GAs with ST converge the second fast while the LC version of GAs have the slowest converge speed. However, since LC's time complexity is positively correlated with the length of parent trees, rather than the graph size, GAs with LC can attain 30 generations while other two GAs attain only one or two generations. This huge advantage makes GAs with LC very easy to take additional measures, e.g., enlarging population sizes and improving the mutation probability, to obtain better solutions. Taking G_7 as an instance, we performed GAs with LC with the population size set to 1000 and the mutation probability set to 0.4. The average execution time per generation in the first few generations is 18.964s, which is also less than that of DT and ST versions of GAs in Table VI. Fig. 6(d) depicts the best weight of each generation and we can see that GAs with LC obtain better performance than that recorded in Fig. 6(a).

VIII. CONCLUSION

In GAs, the crossover operator always occurs with a much larger probability than the mutation operator, thereby playing a very important role in the performance of GAs. This paper proposed a new crossover mechanism called Leaf Crossover (LC) in the context of tree-based GAs. The new crossover mechanism can always generate legal offspring by just exchanging partial parent chromosomes and outperforms existing popular crossover mechanisms by avoiding encoding/decoding operations, the shortest path algorithms, detection procedures, repair functions and global link information. To our best knowledge, LC is the first crossover mechanism without greedy algorithms for Steiner tree based GAs, leading to a significant reduction of the computational time. However, since the repair function has a side effect in enlarging the mutation probability, avoiding it makes GAs with LC behave poorly when the population size is too small to adapt to some complicated problems. Various experiments were conducted to investigate the behavior of GAs with LC. Experimental results show that our GAs can get a better solution with a significant reduction in the time expense. Hence, the new GA can be widely applied to practical problems, especially to networks of large scale and time-urging applications.

For the future work, we would like to study GAs with LC further in practical applications, such as multilayer obstacle-avoiding X-architecture SMT [37], multicast communication and transportation route design. In practical applications, a good GA is often appropriately modified, e.g., a particular mutation operator is adopted for multicast in wireless sensor networks [38]. We would study the adaptive adjustment of

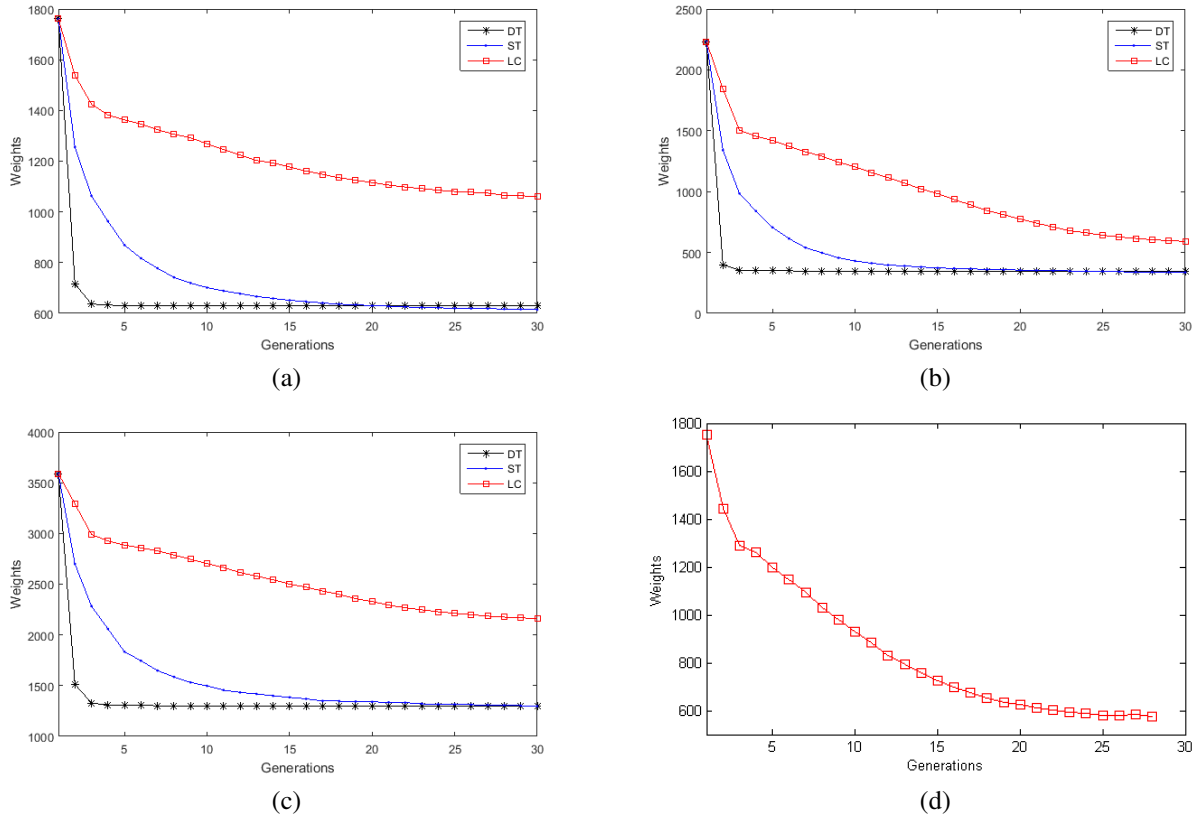


Fig. 6. The best weights in each generation of GAs on large scale graphs. (a) Results on G_7. (b) Results on G_8. (c) Results on G_9. (d) Results of LC with larger populations on G_7.

parameters to meet the requirements of practical applications.

REFERENCES

[1] Y. Yoon and Y.-H. Kim, "An efficient genetic algorithm for maximum coverage deployment in wireless sensor networks," *IEEE Trans. Cybern.*, vol. 43, no. 5, pp. 1473–1483, May 2013.

[2] K. Hadidi, "High-speed general purpose genetic algorithm processor," *IEEE Trans. Cybern.*, vol. 46, no. 7, pp. 1551–1565, Jul. 2016.

[3] C. Perfecto, M. N. Bilbao, J. D. Ser, and A. Ferro, "A simulation-based quantitative analysis on the topological heritability of dandelion-encoded meta-heuristics for tree optimization problems," *Soft Comput.*, vol. 21, no. 17, pp. 4939–4952, Sept. 2017.

[4] F. Rojas and F. Meza, "A parallel distributed genetic algorithm for the prize collecting steiner tree problem," in *Proc. Int. Conf. Comput. Sci. and Comput. Intell.*, 2015, pp. 643–646.

[5] K. Shi, Q. Song, S. Lin, G. Xu, and Z. Cao, "An improved genetic algorithm for degree constrained minimum spanning trees," in *Proc. Control and Decision Conf.*, 2016, pp. 4603–4607.

[6] M. Gen and A. Syarif, "Hybrid genetic algorithm for multi-time period production/distribution planning," *Comput. Ind. Eng.*, vol. 48, no. 4, pp. 799–809, 2005.

[7] Y.-S. Yen, Y.-K. Chan, J. H. Park, and J. H. Park, "A genetic algorithm for energy-efficient based multicast routing on manets," *Comput. Commun.*, vol. 31, no. 10, pp. 2632–2641, 2008.

[8] C. H. Lin and C. C. Chuang, "A rough penalty genetic algorithm for multicast routing in mobile ad hoc networks," *J. Appl. Math.*, vol. 2013, Art. no. 986985, 11 pages, Aug. 2013.

[9] J. Akbari Torkestani and M. R. Meybodi, "A link stability-based multicast routing protocol for wireless mobile ad hoc networks," *J. Netw. Comput. Appl.*, vol. 34, no. 4, pp. 1429–1440, 2011.

[10] T. Lu and J. Zhu, "Genetic algorithm for energy-efficient qos multicast routing," *IEEE Commun. Lett.*, vol. 17, no. 1, pp. 31–34, 2013.

[11] N. Papanna, A. R. M. Reddy, and M. Seetha, "Eelam: Energy efficient lifetime aware multicast route selection for mobile ad hoc networks," *Appl. Comput. Inform.*, 2017.

[12] J. Knowles and D. Corne, "A new evolutionary approach to the degree-constrained minimum spanning tree problem," *IEEE Trans. Evol. Comput.*, vol. 4, no. 2, pp. 125–134, 2000.

[13] T. Lu, J. Zhu, S. Chang, and L. Zhu, "Maximizing multicast lifetime in unreliable wireless ad hoc network," *Wireless Netw.*, vol. 24, pp. 1175-1185, Nov., 2016.

[14] P. C. Pop, O. Matei, C. Sabo, A. Petrovan, P. C. Pop, O. Matei, C. Sabo, and A. Petrovan, "A two-level solution approach for solving the generalized minimum spanning tree problem," *Europ. J. Oper. Res.*, vol. 265, no. 2, pp. 478–487, Mar. 2018.

[15] X. Gao and L. Jia, "Degree-constrained minimum spanning tree problem with uncertain edge weights," *Appl. Soft Comput.*, vol. 56, pp. 580–588, Jul. 2016.

[16] A. T. Haghighat, K. Faez, M. Dehghan, A. Mowlaei, and Y. Ghahremani, "A genetic algorithm for steiner tree optimization with multiple constraints using prufer number," in *Proc. Inf. Commun. Technol. EurAsia-ICT*, 2002, pp. 272–280.

[17] S. Picciotto, "How to encode a tree," *arXiv preprint arXiv:1710.08463*, 2017.

[18] S. Caminiti and R. Petreschi, "String Coding of Trees with Locality and Heritability," in *Proc. 2005 Comput. Comb. Conf.*, 2005, pp. 251-262.

[19] T. W. D. Lima, A. C. B. Delbem, A. D. S. Soares, F. M. Federson, J. B. A. L. Junior, and J. V. Baalen, "Node-depth phylogenetic-based encoding, a spanning-tree representation for evolutionary algorithms. Part I: Proposal and properties analysis," *Swarm Evol. Comput.*, vol. 31, pp. 1–10, Dec. 2016.

[20] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Labs Tech. J.*, vol. 36, no. 6, pp. 1389–1401, 2013.

[21] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and R. Talebi, "On-demand location aware multicast (olam) for ad hoc networks," in *Proc. 2000 IEEE Wireless Commun. Netw. Conf.*, vol. 3, 2000, pp. 1323–1328.

[22] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[23] V. K. Dabhi and S. Chaudhary, "Empirical modeling using genetic programming: A survey of issues and approaches," *Nat. Comput.*, vol. 14, no. 2, pp. 303–330, 2015.

[24] W. B. Langdon, "Size fair and homologous tree crossovers for tree

genetic programming," *Genet. Progr. Evol. Mach.*, vol. 1, no. 1-2, pp. 95–119, 2000.
[25] M. Šprogar, "Prudent alignment and crossover of decision trees in genetic programming," *Genet. Progr. Evol. Mach.*, vol. 16, no. 4, pp. 499–530, 2015.
[26] H. Jabeen and A. R. Baig, "Depthlimited crossover in GP for classifier evolution," *Comput. Human Behav.*, vol. 27, no. 5, pp. 1475–1481, 2011.
[27] A. Diveev, "Small variations of basic solution method for non-numerical optimization," *IFAC PapersOnLine*, vol. 48, no. 25, pp. 28–33, 2015.
[28] Y.-S. Yen, H.-C. Chao, R.-S. Chang, and A. Vasilakos, "Flooding-limited and multi-constrained QoS multicast routing based on the genetic algorithm for MANETs," *Math. Comput. Model.*, vol. 53, no. 11, pp. 2238–2250, 2011.
[29] J. Zhou, X. Yi, K. Wang, and J. Liu, "Uncertain distribution-minimum spanning tree problem," *Int. J. Uncertainty, Fuzziness Knowl. Syst.*, vol. 24, no. 04, pp. 537–560, 2016.
[30] U. Mehboob, J. Qadir, S. Ali, and A. Vasilakos, "Genetic algorithms in wireless networking: techniques, applications, and issues," *Soft Comput.*, vol. 20, no. 6, pp. 2467–2501, 2016.
[31] P. Karthikeyan and S. Baskar, "Genetic algorithm with ensemble of immigrant strategies for multicast routing in ad hoc networks," *Soft Comput.*, vol. 19, no. 2, pp. 489–498, 2015.
[32] T. Paulden and D. K. Smith, "From the dandelion code to the rainbow code: a class of bijective spanning tree representations with linear complexity and bounded locality," *IEEE Trans. Evol. Comput.*, vol. 10, no. 2, pp. 108–123, 2006.
[33] G. Zhout and M. Gen, "An effective genetic algorithm approach to the quadratic minimum spanning tree problem," *Comput. Oper. Res.*, vol. 25, no. 3, pp. 229–237, 1998.
[34] Z. Wang, B. Shi, and E. Zhao, "Bandwidth-delay-constrained least-cost multicast routing based on heuristic genetic algorithm," *Comput. Commun.*, vol. 24, no. 7, pp. 685–692, 2001.
[35] X. Hue, "Genetic algorithms for optimization: Background and applications," *Edinburgh Parallel Computing Centre*, vol. 10, 1997.
[36] S. Oh, C. W. Ahn, and R. S. Ramakrishna, "A genetic-inspired multicast routing optimization algorithm with bandwidth and end-to-end delay constraints," in *Proc. Neural Inf. Process.*, 2006, pp. 807–816.
[37] G. Chen, "Multilayer obstacle-avoiding x-architecture steiner minimal tree construction based on particle swarm optimization," *IEEE Trans. Cybern.*, vol. 45, no. 5, pp. 989–1002, 2015.
[38] J. Zhou, Q. Cao, C. Li, and R. Huang, "A genetic algorithm based on extended sequence and topology encoding for the multicast protocol in two-tiered WSN," *Expert Syst. Appl.*, vol. 37, no. 2, pp. 1684–1695, 2010.

Qiongbing Zhang received his BSc degree in electronic and information engineering from Central South University, Changsha, China, in 2009, and MSc and PhD degrees from the State Key Laboratory of Software Engineering (SKLSE), Wuhan University, Wuhan, China, in 2011 and 2017 respectively.

He is currently a lecturer at the school of Computer Science and Engineering, Hunan University of Science and Technology, China. His research interests include intelligence computation and intelligent information processing.

Shengxiang Yang (M'00–SM'14) received the PhD degree from Northeastern University, Shenyang, China in 1999.

He is currently a Professor in Computational Intelligence and Director of the Centre for Computational Intelligence, School of Computer Science and Informatics, De Montfort University, Leicester, U.K. He has over 300 publications with over 10500 citations and an H-index of 55 according to Google Scholar. His current research interests include evolutionary computation, swarm intelligence, artificial neural networks, data mining and data stream mining, and relevant real-world applications.

Prof. Yang serves as an Associate Editor/Editorial Board Member of a number of international journals, such as the *IEEE Transactions on Evolutionary Computation*, *IEEE Transactions on Cybernetics*, *Information Sciences*, and *Enterprise Information Systems*.

Min Liu received the MSc degree in computer application technology from Xiangtan University, Xiangtan, China, in 2006, and the PhD degree in foundation of artificial intelligence from Xiamen University, Xiamen, China in 2012.

From 2006 to 2013, he was a Lecturer with the School of Computer, Minnan Normal University, Zhangzhou, China. Since 2013, he has been a Lecturer with the School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan, China. His research interests include evolutionary computation, multiobjective optimization, and dynamic optimization.

Jianxun Liu received his MSc and PhD degrees in computer science from the Central South University of Technology in 1997 and Shanghai Jiao Tong University in 2003, respectively.

He is currently a professor and dean in the School of Computer Science and Engineering, Hunan University of Science and Technology. His current interests include service computing and cloud computing, BPM and Big Data, etc. He has published more than 60 papers in peer-reviewed international journals and conferences.

Lei Jiang received his BSc and MSc degrees in Computer Science from the Southwest Petroleum University in 1996 and 2005, respectively, and PhD degree from the State Key Laboratory of Software Engineering (SKLSE), Wuhan University, Wuhan, China, in 2012.

He is currently an associate professor of Department of Computer Science and Engineering, Hunan University of Science and Technology in Xiangtan, China. His research interests include social computing, evolutionary computation, machine learning and decision support.

A New Crossover Mechanism for Genetic Algorithms for Steiner Tree Optimization – Supplemental Material

Qiongbing Zhang, Shengxiang Yang, Min Liu, Jianxun Liu, and Lei Jiang

This is a supplementary material of the paper "A New Crossover Mechanism for Genetic Algorithms for Steiner Tree Optimization", published in IEEE Transactions on Cybernetics (DOI: 10.1109/TCYB.2020.3005047).

I. The detailed description of leaf crossover (LC)

For convenience, the notations is outlined as follow.

Leaf(T)=$\{v|$leaf nodes in tree $T\}$.

Child(E)=$\{v|$the child node of $link(v,u)$, $link(v,u) \in \mathbf{E}\}$.

$|$**Child(v)**$|$: the number of child nodes of the node v.

Connect$(T_1, newT, v)$= $\{Path(v,\mu)|Path(v,\mu) \subseteq T_1$, $degree(T_1,\mu)$>$degree(T_1,v)$, $\mu \in \mathbf{C}hild(E_{newT})$ and $\{V_p - \mu\} \cap Child(E_{newT}) = \phi\}$.

Hop(v,u,T): the number of hops between v and u in tree T.

F'(T)= $\omega_0 - \omega(T)$.

Cut(v,u,P)= $Path(v,u)$. For any path P, V_p is set of the nodes in P, E_p is set of links in P and $Cut(v,u,P)$ denotes the path by removing all edges of P expect edges connecting v to u, where $v \in \mathbf{V}_p$, $u \in \mathbf{V}_p$.

Removing Rules. For a path $P \subset T$, the rules for a parent tree T to remove the path P is listed as follows.

1) If P is chosen to be added into the new tree, then T removes all edges in P. Furthermore, if node $v \in \mathbf{T}argets \cap Child(E_p)$, set $Targets = Targets - v$;

2) If P is not added into the new tree, remove it from T as follows.

Step 1: $v = Leaf(P)$;

Step 2: $T = T - v$, $P = P - v$;

Step 3: $v = Leaf(P)$, if $v \in \mathbf{T}argets$ or $|child(v)|$>1, stop the iteration; otherwise, go to Step 2. Furthermore, if $v \in \mathbf{T}argets$, set $leaf = leaf + v$.

Fig. 7(a-f) presents the overall operation of the LC crossover on two tree instances in Fig. 2 of the paper. We firstly obtain the subtree set $newT = \{V(E_1 \cap E_2), E_1 \cap E_2\}$. This step is analogous to the first procedure in the crossover of GAs with DT. The two chromosomes chosen for crossover select the same links and copy them to the child directly, except that this procedure does not copy target nodes to the child and we add a link $(s,-1)$ for the root node s. As Fig. 7(a) shows, the generated new tree is $newT = \{V(E_1 \cap E_2)$, $E_1 \cap E_2$=$\{\{-1,1,2,14,15\},\{(1,-1),(2,1),(15,14)\}\}$.

Meanwhile, leaf node lists of T_1 and T_2: $list_1 = \{3,9,14\}$ and $list_2 = \{3,9,14\}$, are generated after the procedure: $T_1 - newT$ and $T_2 - newT$. Seeing that $newT$ contains node 15, the remaining target nodes are $Targets = \{3,6,7,9\}$.

Then, we randomly select a node $v \in \mathbf{l}eaf_1 \cap leaf_2$ to obtain two paths: $P_1 = Connect(T_1, newT, v)$ and $P_2 = Connect(T_2, newT, v)$. The next operation is to find whether $\{P_1 \cap P_2 - v\} = \phi$. If not, then we select the node $u = min\{Hop(v,u,T_1) + Hop(v,u,T_2)|u \in \{P_1 \cap P_2 - v\}\}$ and cut off P_1 and P_2: $P_1 = Cut(v,u,P_1)$ and $P_2 = Cut(v,u,P_2)$. Meanwhile, $leaf_1 = \{leaf_1 + u\}$ and $leaf_2 = \{leaf_2 + u\}$. As Fig. 7(b) shows, when $v = 3$, $P_1 = Connect(T_1, newT, 3)$=$Path(3,1)$=$\{(3,1)\}$ and $P_2 = Connect(T_1, newT, 3) = Path(3,1) = \{(3,12), (12,6),(6,7),(7,5),(5,2)\}$. Given $\{P_1 \cap P_2 - 3\} = \phi$, we do not need to cut two paths off.

Next, we check if a loop exists in $newT \cup P_1$, if a loop exists, then $P_1 = \phi$. We perform the same operation to P_2.

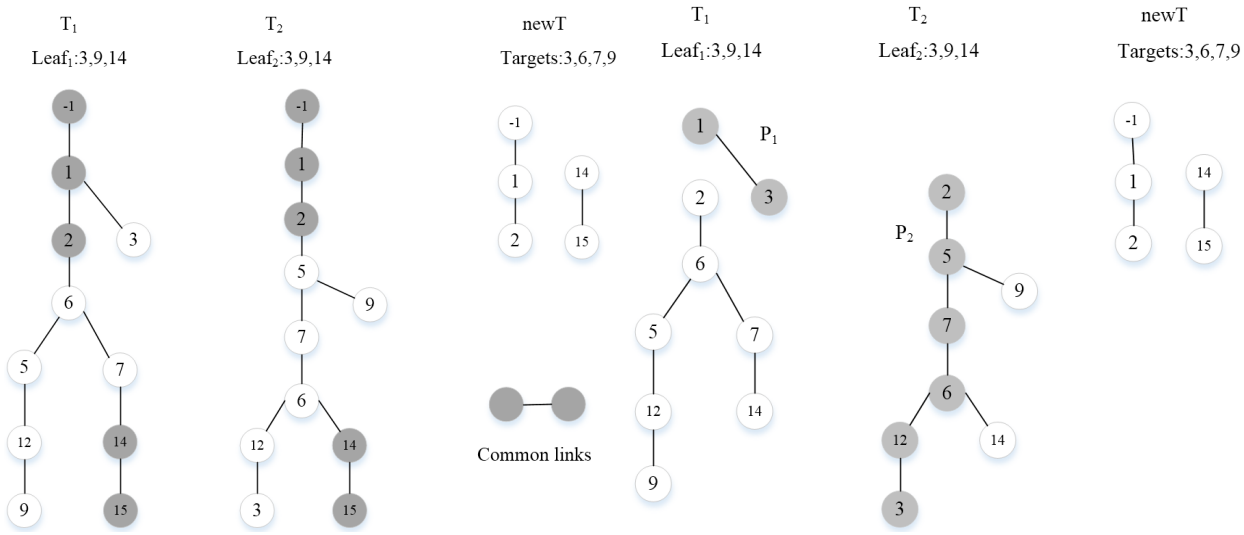
Finally, we compare $F'(P_1)$ with $F'(P_2)$ and add the larger path into $newT$. According to the *Remove Rules*, we do the removing procedures: $T_1 - P_1$ and $T_2 - P_2$. Fig. 1(c) illustrates the case of $F'(P_1) > F'(P_2)$, where P_1 is added into $newT$. Meanwhile, the removal procedure removes P_1, P_2 from T_1, T_2 according to *Remove Rules*. We remove all edges of P_1 from T_1 and remove from T_2 nodes from the leaf nodes of P_2 to top by the iteration procedure. The iterations are stopped on node 6 given that $|child(6)| > 1$. Seeing that node $6 \in \mathbf{T}argets$, it follows that $leaf_2 = \{leaf_2$+6$\} = \{6,9,14\}$.

Fig. 7(c) depicts the next iteration of leaf crossover when the chosen node in $leaf_1 \cap leaf_2$ is 9. $P_1 = Connect(T_1, newT, 9) = Path(9,2) = \{(9,12),(12,5),(5,6),(6,2)\}$ and $P_2 = Connect(T_2, newT, 9) = Path(9,2) = \{(9,5),(5,2)\}$. However, two paths should be cut off at node 5 given that node $5 = min\{Hop(9,u,T_1) + Hop(9,u,T_2)|u \in \{P_1 \cap P_2 - 9\}\}$. Hence, $P_1 = Cut(9,5,P_1) = \{(9,12),(12,5)\}$, $P_2 = Cut(9,5,P_2) = \{(9,5)\}$. Meanwhile, both $list_1$ and $list_2$ add node 5, $leaf_1 = \{leaf_1 + 5\}$, $leaf_2 = \{leaf_2 + 5\}$. No loop exists in either $newT \cup P_1$ or $newT \cup P_2$. Assuming $F'(P_2) > F'(P_1)$, then $newT = newT + P_2$ and we remove P_1, P_2 from parent trees T_1, T_2 according to *Remove Rules*.

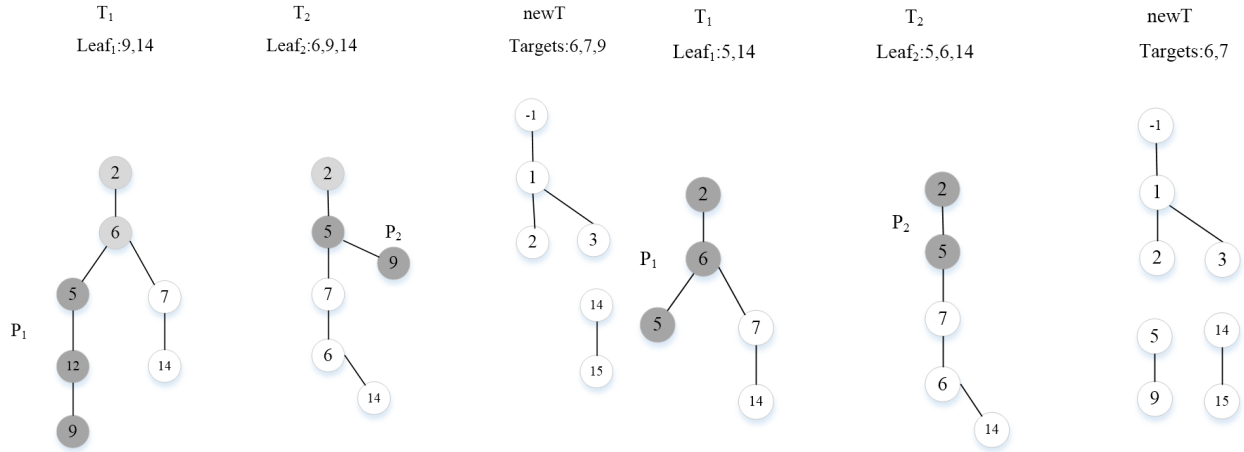
Q. Zhang is with the School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan 411201, China e-mail: mrtly2@whu.edu.cn.

S. Yang is with the School of Computer Science and Informatics, De Montfort University, Leicester LE1 9BH, U.K.

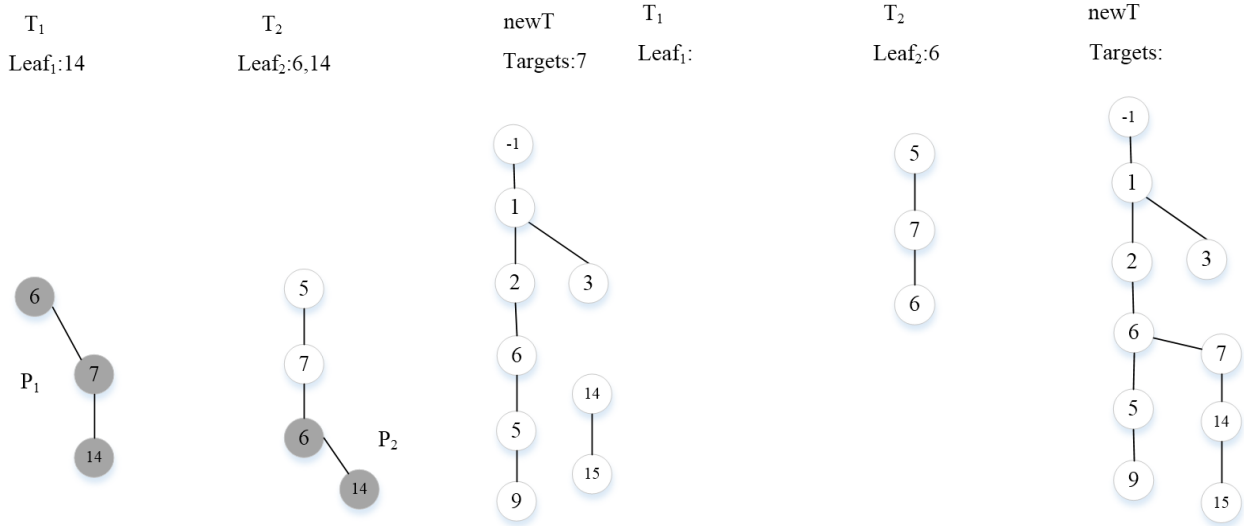
M. Liu, J. Liu and L. Jiang are with the School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan 411201, China.



(a) *Step 1 of leaf crossover.* (b) *The selected same leaf node is* 3.



(c) *The selected same leaf node is*9. (d) *The selected same leaf node is*5.



(e) *The selected same leaf node is*14. (f) *The offspring tree.*

Fig. 7. Leaf crossover on T_1 and T_2.

Fig. 7(d) presents the next iteration of $P_1 = Connect(T_1, newT, 5) = \{(5,6),(6,2)\}$, $P_2 = Connect(T_2, newT, 5) = \{(5,2)\}$ when the chosen node in $leaf_1 \cap leaf_2$ is 5. Assuming $F'(P_1) > F'(P_2)$, we set $newT = newT + P_1$ and remove P_1, P_2 from T_1, T_2 according to ***Remove Rules***.

Fig. 7(e) reveals that the last node in $leaf_1 \cap leaf_2$ is 14, $P_1 = Connect(T_1, newT, 14) = \{(14,7),(7,6)\}$, $P_2 = Connect(T_2, newT, 14)$=$\{(14,6),(6,7),(7,5)\}$. Noting that both node 6 and node 7 achieve the minimum in $min\{Hop(14,u,T_1) + Hop(14,u,T_2) | u \in \{P_1 \cap P_1 - 14\}\}$. Assuming that node 6 is the selected node, we get $P_1 = Cut(14,6,P1) = \{(14,7),(7,6)\}$, $P_2 = Cut(14,6,P_2) = \{(14,6)\}$. Under the condition $F'(P_1) > F'(P_2)$, we add P_1 into $newT$ and remove P_1, P_2 from T_1, T_2 respectively according to ***Remove Rules***.

The iterations cannot proceed when $leaf_1 \cap leaf_2 = \phi$. If $T_1 = \phi$ or $T_2 = \phi$ at this moment, then the crossover is over. Otherwise, we compare $F'(T_1)$ with $F'(T_2)$ and add the larger tree into $newT$. It can be seen from Fig. 7(f) that T_1 satisfies the condition of being an empty tree, so the crossover is over and $newT$ is the final offspring tree.