CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	A constraint-based parallel local search for the edge-disjoint rooted distance-constrained minimum spanning tree problem
Authors	Arbelaez, Alejandro;Mehta, Deepak;O'Sullivan, Barry;Quesada, Luis
Publication date	2017-06-16
Original Citation	Arbelaez, A., Mehta, D., O'Sullivan, B. and Quesada, L. (2017) 'A constraint-based parallel local search for the edge-disjoint rooted distance-constrained minimum spanning tree problem', Journal of Heuristics, 24, pp. 359-394. https://doi.org/10.1007/s10732-017-9342-0
Type of publication	Article (peer-reviewed)
Link to publisher's version	10.1007/s10732-017-9342-0
Rights	© 2017, Springer Science+Business Media, LLC. This is a post-peer-review, pre-copyedit version of an article published in Journal of Heuristics. The final authenticated version is available online at: https://doi.org/10.1007/s10732-017-9342-0
Download date	2026-11-04 11:32:56
Item downloaded from	https://hdl.handle.net/10468/16671



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

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

A Constraint-based Parallel Local Search for Edge Disjoint Rooted Distance-Constrained Minimum Spanning Tree Problem

Alejandro Arbelaez · Deepak Mehta · Barry O'Sullivan · Luis Quesada

Received: date / Accepted: date

Abstract Many network design problems arising in areas as diverse as VLSI circuit design, QoS routing, traffic engineering, and computational sustainability require clients to be connected to a facility under path-length constraints and budget limits. These problems can be seen as instances of the Rooted Distance-Constrained Minimum Spanning-Tree problem (RDCMST), which is NP-hard. An inherent feature of these networks is that they are vulnerable to a failure. Therefore, it is often important to ensure that all clients are connected to two or more facilities via edge-disjoint paths. We call this problem the Edge-disjoint RDCMST (ERDCMST). Previous works on RDCMST have focused on dedicated algorithms and therefore it is difficult to use these algorithms to tackle ERDCMST. We present a constraint-based parallel local search algorithm for solving ERDCMST. Traditional ways of extending a sequential algorithm to run in parallel perform either portfolio-based search in parallel or parallel neighbourhood search. We rather exploit the semantics of the constraints of the problem to perform multiple moves in parallel by ensuring that they are mutually independent. The ideas presented in this paper are general and can be adapted to other problems as well. The effectiveness of our approach is demonstrated by experimenting with a set of problem instances

Alejandro Arbelaez
Insight Centre for Data Analytic, University College Cork, Ireland
E-mail: alejandro.arbelaez@insight-centre.org

Deepak Mehta
Insight Centre for Data Analytic, University College Cork, Ireland
E-mail: deepak.mehta@insight-centre.org

Barry O'Sullivan
Insight Centre for Data Analytic, University College Cork, Ireland
E-mail: barry.osullivan@insight-centre.org

Luis Quesada
Insight Centre for Data Analytic, University College Cork, Ireland
E-mail: luis.quesada@insight-centre.org

taken from real-world passive optical network deployments in Ireland, Italy, and the UK. Our results show that performing moves in parallel can significantly reduce the elapsed time and improve the quality of the solutions of our local search approach.

Keywords Local Search · Optical Networks · Parallelism

1 Introduction

Many network design problems arising in areas as diverse as VLSI circuit design, QoS routing, traffic engineering, and computational sustainability require clients to be connected to a facility under path-length constraints and budget limits. Here the length of the path can be interpreted as distance, delay, signal loss, etc. For example, in a multicast communication setting where a single node is broadcasting to a set of clients, it is important to restrict the path delays from the server to each client. In Long-Reach Passive Optical Networks (LR-PON), a metro node (MN) is connected to the a set of exchange-sites via optical fibres, the length of the fibre between an exchange-site (ES) and its metro node is bounded by the signal loss. The goal is to minimise the cost resulting from the total length of fibres [20]. In VLSI circuit design, path delay is a function of the maximum interconnection path length while power consumption is a function of the total interconnection length [19]. In package shipment services, guarantee constraints are expressed as restrictions on total travel time from an origin to a destination, and the organisation wants to minimise the transportation costs [22].

Many of these network design problems are instances of the Rooted Distance-Constrained Minimum Spanning-Tree Problem (RDCMST) [19], which is NP-hard. The objective is to find a minimum cost spanning tree with the additional constraint that the length of the path from a specified root-node (or facility) to any other node (client) must not exceed a given threshold. Many networks are complex systems that are vulnerable to a failure. A major fault occurrence would be a complete failure of the facility, which would affect all the clients connected to the facility. Therefore it is important to provide network resilience. We restrict our attention to the networks where all clients are required to be connected to two facilities via two edge disjoint paths so that whenever a single facility fails or a single link fails all clients are still connected to at least one facility. We define this problem as the Edge-disjoint Rooted Distance-Constrained Minimum Spanning-Trees Problem (ERDCMST). Given a set of facilities and a set of clients such that each client is associated with two facilities, the problem is to find a set of distance-constrained spanning trees rooted from each facility with minimum total cost. Additionally, each client is connected to its two facilities via two edge disjoint paths. This would effectively mean that each pair of distance-bounded spanning trees would be mutually disjoint in terms of edges.

Previous works on RDCMST [16, 23] have focused on dedicated algorithms which are hard to extend with side constraints, and therefore it is difficult

to use these algorithms to tackle ERDCMST. We present a mixed integer programming formulations of ERDCMST and a constraint-based local search algorithm, which can easily be extended to apply widely. We present two efficient local move operators and an incremental way of maintaining the objective function, which is often a key element for the efficiency of a local search algorithm. Our local search algorithm is able to solve both RDCMST and ERDCMST. Although these move operators were proposed and evaluated with respect to edge-disjoint, capacity, and distance constraints in [7] and [5], in this paper we provide a step-by-step illustration of these move operators with more examples, and explain how the incremental data structures are being updated.

We extend our sequential algorithm and propose a parallel version of our constraint-based local search algorithm.[1] The traditional way of extending a sequential algorithm to run in parallel is to perform either portfolio-based search in parallel or parallel neighbourhood search. We rather exploit the semantics of the constraints of the problem to perform multiple moves in parallel by ensuring that they do not conflict with each other. The effectiveness of our approach is demonstrated by experimenting with a set of problem instances taken from real-world passive optical network deployments in Ireland, the UK and Italy. Our results show that performing moves in parallel can significantly reduce the time required to find a target solution and it improves the anytime behaviour of our local search algorithm.

2 Formal Specification and Complexity

Let G be a directed graph with set of nodes $\mathcal{N}$ and set of edges $\mathcal{L}$. We assume that G is a complete graph without self-loops, so $|\mathcal{L}| = |\mathcal{N}| \times (|\mathcal{N}| - 1)$ in our case. Each edge has a cost and a distance value associated with it. Let $\mathcal{M}$ be a set of facilities and let $u_i \in \mathcal{U}$ be a set of (users or) clients. We define $\mathcal{N}$ as the union of $\mathcal{M}$ and $\mathcal{U}$. Let $U_i \subseteq \mathcal{U}$ be the set of clients that are associated with facility $m_i \in \mathcal{M}$. We use T_i to denote the tree network associated with facility i. We also use $N_i = U_i \cup \{m_i\}$ to denote the set of nodes in the tree T_i associated with the facility m_i, and a set of edges $L_i \subseteq N_i^2$

T_i is a subgraph of G that contains a directed path from facility m_i to all of its clients and contains no cycles. The length of a path (or path-length) between two nodes is the sum of the distances of the edges connecting the two nodes, and the cost of T_i is the sum of the cost of its edges. In this paper, without loss of generality, we assume that the cost is symmetrical, i.e., the cost of an edge $\langle u_p, u_q \rangle$ is equal to the cost of the edge $\langle u_q, u_p \rangle$. As mentioned before, we also assume that the graph is complete since non-existing edges in the original graph can be represented by edges with a very large distance with respect to the distance threshold.

[1] Preliminary results of the proposed algorithm have been presented in a workshop without formal proceedings [6].

Rooted Minimum Spanning Tree Problem (RMST). Given a graph $\mathcal{G} = (N_i, L_i)$ with a facility $m_i \in \mathcal{M}$ and a real value c_{jk} denoting the cost of each edge $\langle u_j, u_k \rangle \in L_i$, RMST is to find a spanning tree of minimum cost of $\mathcal{G}$.

Rooted Distance-Constrained Minimum Spanning-Tree Problem (RDCMST). Given a graph $\mathcal{G} = (N_i, L_i)$ with a facility $m_i \in \mathcal{M}$, the set of clients U_i, two real values c_{jk} and d_{jk} denoting the cost and the distance of each edge $\langle u_j, u_k \rangle \in L_i$, and a real value λ, RDCMST is to find a spanning tree T_i with minimum cost of $\mathcal{G}$ such that the length of the path from the facility m_i to any client $u_j \in U_i$ is not greater than λ.

Edge-Disjoint Rooted Distance-Constrained Minimum Spanning-Trees Problem (ERDCMST). Given a graph $\mathcal{G}=(\mathcal{N}, \mathcal{L})$ with a set of facilities $\mathcal{M}$, a set of clients $\mathcal{U}$, a set edges $\mathcal{L}$, two real values c_{jk}, and d_{jk} denoting the cost and distance of each edge $\langle u_j, u_k \rangle \in \mathcal{L}$, an association of clients with two facilities $\pi : \mathcal{U} \rightarrow \mathcal{M}^2$, and a real value λ, ERDCMST is to find a spanning tree T_i of $\mathcal{G}$ for each facility m_i such that:

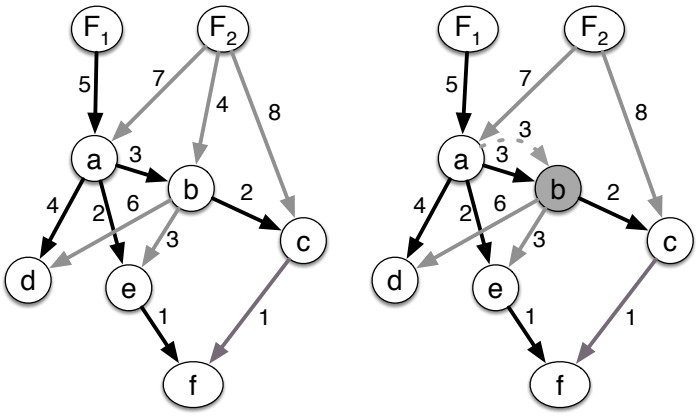
1. The length of the unique path for all m_i to any of its clients is not greater than λ.
2. For each client u_k, the two paths connecting u_k to m_i and m_j, where $\pi(u_k) = \langle m_i, m_j \rangle$, are edge disjoint.
3. The sum of the costs of the edges in all the spanning trees is minimum.

Figure 1 shows an example with two facilities F_1 and F_2 and N=$\{a, b, c, d, e, f\}$, black (respectively grey) edges denote the set of edges used to reach F_1 (respectively F_2), the value for λ is 16 and the total cost of the solution is 46 for this illustrative example. Figure 1(a) shows a valid solution satisfying both distance and edge-disjointness constraints. The distance from the facilities to any node is less than or equal to $\lambda = 16$. The paths connecting the set of nodes to the facilities are edge disjoint. Figure 1(b) shows a cheaper solution replacing a grey edge $\langle F_2, b \rangle$ with $\langle a, b \rangle$, but this solution would not be edge disjoint since a failure in $\langle a, b \rangle$ would disconnect node b from the two facilities.

Complexity. ERDCMST involves finding a rooted distance-bounded spanning tree for every facility whose total cost is minimum. This problem is known to be NP-hard [19].

3 A mathematical model for ERDCMST

In this section we present a mixed integer programming formulation of ERDCMST.



(a) Satisfying both distance and edge disjointness

(b) Satisfying distance constraint but violating edge disjointness

Fig. 1 Example of an instance of the ERDCMST problem. λ=16

Variables:

- Let x^i_{jk} be a Boolean variable that denotes whether an edge from node $u_j \in N_i$ to $u_k \in N_i$ of facility $m_i \in \mathcal{M}$ is selected or not.
- Let f^i_j be a variable that denotes the upper bound on the length of the path from the facility m_i to its client u_j.

We remark that the partial order enforced by f helps to rule out cycles in the solution.

Constraints: Each facility is directly connected to at least one of its clients:

$$\forall_{m_i \in \mathcal{M}} : \quad \sum_{u_j \in U_i} x^i_{ij} \geq 1$$

The total number of edges in any tree T_i is equal to $|U_i|$:

$$\forall_{m_i \in \mathcal{M}} : \quad \sum_{u_j \in N_i} \sum_{u_k \in U_i, u_j \neq u_k} x^i_{jk} = |U_i|$$

the distance from the root node (or facility) to itself is 0.

$$\forall_{m_i \in \mathcal{M}} : f^i_i = 0$$

The length of the path from any client to its facility is bounded by λ:

$$\forall_{m_i \in \mathcal{M}} \forall_{u_j \in N_i} : \quad f^i_j \leq \lambda$$

If there is an edge from $u_j \in U_i$ to $u_k \in U_i$ then the length of the path from m_i to u_k is equal to the sum of the length of the path from m_i to u_j

plus the distance between u_j and u_k. We use the big-M method to model this implication as a linear constraint. The value of the constant $\mathcal{C}$ has to be greater than λ plus the maximum distance between any pair of edges in order to be consistent with the implication.

$$\forall_{m_i \in \mathcal{M}} \forall_{\{u_j,u_k\} \in N_i} : \quad \mathcal{C}(1 - x^i_{jk}) + f^i_k \geq f^i_j + d_{jk}$$

If m_i and $m_{i'}$ are the facilities of the client j, and if there exists any path in the subnetwork associated with facility i that includes the edge $\langle u_j, u_k \rangle$, then facility i' cannot use the same edge. Therefore, we enforce the following constraint:

$$\forall_{\{m_i,m_{i'}\} \in \mathcal{M}} \forall_{\{u_j,u_k\} \in U_i \cap U_{i'}} : x^i_{jk} + x^{i'}_{jk} \leq 1$$

Objective. The objective is to minimise the total cost:

$$\min \sum_{m_i \in \mathcal{M}} \sum_{\{u_j,u_k\} \in N_i} c_{jk} \cdot x^i_{jk}$$

4 Iterated Constraint-based Local Search

For large networks containing more than 10000 clients and 100 facilities it is unexpected to solve the previously presented models using systematic search. In this section we present an iterated constraint-based local search approach for solving our problem.

The Iterated Constraint-based Local Search (ICBLS) [14,27] framework depicted in Algorithm 1 comprises two phases. First, in a local search phase, the algorithm improves the current solution, little by little, by performing small changes. The algorithm employs a move operator in order to move from one solution to another in the hope of improving the value of the objective function. Second, in the perturbation phase, the algorithm perturbs the incumbent solution (s^*) in order to escape from difficult regions of the search (e.g., a local minima). The acceptance criterion decides whether to update s^* or not. The algorithm accepts s'^* with a probability p, and s^* in the other cases. Furthermore, we limit the space of candidate solutions to valid solutions satisfying the constraints of the problem.

Algorithm 1 Iterated Constraint-Based Local Search(*move-op*, s)

1: s^* := ConstraintBasedLocalSearch(*move-op*, s)
2: **repeat**
3: s' := Perturbation(s^*)
4: s'^* := ConstraintBasedLocalSearch(*move-op*, s')
5: s^* := AcceptanceCriterion(s^*, s'^*)
6: **until** a given stopping criterion is met

Our algorithm requires two parameters: s the initial solution where all clients are able to reach their facilities while satisfying all constraints (i.e., the

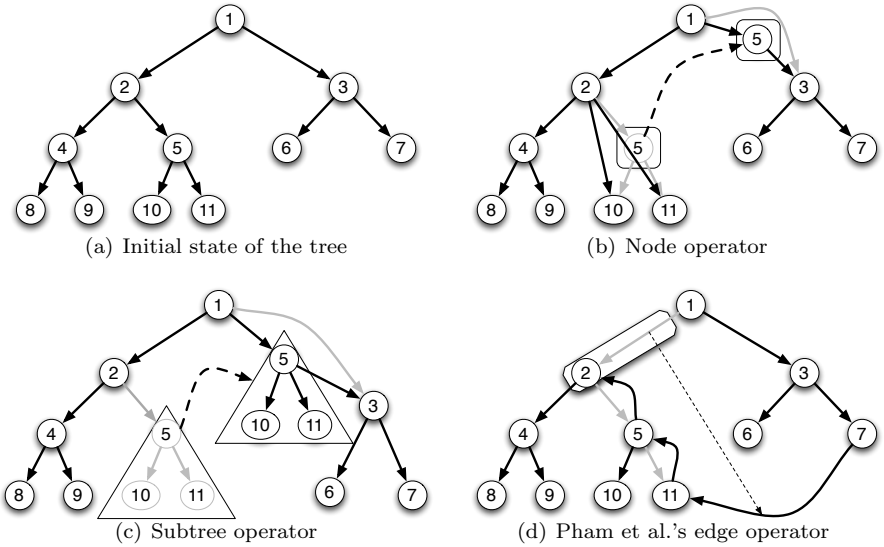


Fig. 2 Move operators. Grey arrows indicate edges removed from the current solution.

upper bound in the length and disjointness), and the move operation (*move-op*) which is a function itself. We switch from the local search phase to the perturbation phase when a local minima is observed. In the perturbation phase we perform a given number of random moves. In this paper, unless otherwise stated, we use the trivial solution of connecting all clients directly to their respective facilities as the initial solution. The stopping criteria is either a timeout or a given number of iterations.

4.1 Move operators

In this section we describe the *node*, *subtree*, and *edge* move operators. We use T_i to denote the tree associated with facility i. An edge between two clients u_j and u_k is denoted by $\langle u_j, u_k \rangle$. We have defined *node* and *subtree* in [7], and we took the `edge` operator from the literature [12]. Informally speaking the location of a node u_j (or subtree emanating from it) in a tree is a tuple (u_{p_j}, S_j) where u_{p_j} denotes the predecessor of u_j and S_j denotes the list of immediate successors of u_j in the tree.

Node operator (Figure 2(b)) moves a given node u_i from the current location to another in the tree. The node (or subtree) is changing location if either the predecessor or any successor of the node is different. As a result of the move, all successors of u_i will be directly connected to the predecessor node of u_i. u_i can be placed as a new successor for another node or in the middle of an existing edge in the tree.

Subtree operator (Figure 2(c)) moves a given node u_i and the subtree emanating from u_i from the current location to another in the tree. As a result of this, the predecessor of u_i is no longer connected to u_i, and all successors of u_i are still directly connected to u_i. u_i can be placed as a new successor for another node or in the middle of an existing edge.

Edge Operator (Figure2(d)). In this paper we limit our attention to moving a node or a complete subtree. In [12] the authors proposed to move edges in the context of the Constrained Optimum Path problem. Pham et al. move operator (Figure 2(d)) chooses an edge in the tree and finds another location for it without breaking the flow.

4.2 Operations and Complexities

We first present the complexities of the node and subtree operators as they share similar features. For an efficient implementation of the move operators, it is necessary to maintain b^i_j: the length of the path from u_j down the the farthest leaf associated with the tree T_i, and the previously described variable f^i_j: the distance from the facility to the client. Let u_{p_j} be the immediate predecessor of u_j and let S_j be the set of immediate successors of u_j in a given tree.

In order to complete a move the node and subtree operators require to execute the following four steps:

1. Randomly select a node (u_j) from a facility (m_i) from the current solution;
2. Delete u_j of T_i if the node operator is used, or u_j and the emanating subtree of the node in T_i if the subtree operator is used;
3. Identify the best location, i.e., a new predecessor u_{p_j} and a potential new successor u_{ns_j} for u_j in T_i satisfying all constraints;
4. Insert u_j as a new successor of u_{p_j}, and if there is a new potential successor, add u_{ns_j} as a new predecessor of u_p.

Table 1 summarises the complexities of the move operators. In this table n denotes the maximum number of clients associated with a single facility. The last row (move) indicates the time complexity of completing a move with a given move operator, i.e., completing the four previously mentioned steps.

In Figure 3 we show an initial solution, which is used to illustrate the operations and how the incremental data structures are being updated. Initially we provide the complexity analysis only for the distance constraint and later on we show that supporting disjointness require the same overall time complexity. Each edge is assigned a weight representing the distance of traversing the edge. The total cost of the solution is the sum of the weights of the edges in the tree.

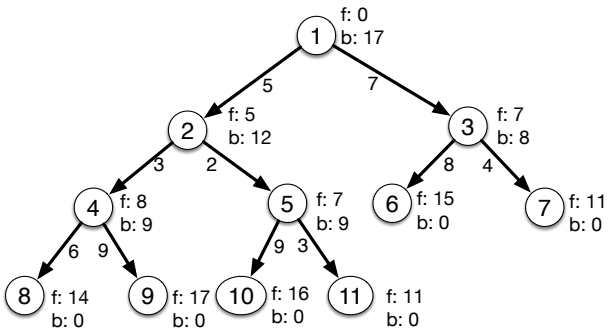


Fig. 3 Initial Tree. Edges are labeled with the cost of connecting the two nodes. The total cost of this solution is 56.

Table 1 Complexities of different operations

	Node	Subtree	Edge
Delete	$O(n)$	$O(n)$	$O(1)$
Feasible delete	$O(n)$	$O(1)$	$O(1)$
Feasible insert	$O(1)$	$O(1)$	$O(n)$
Best location	$O(n)$	$O(n)$	$O(n^3)$
Insert	$O(n)$	$O(n)$	$O(n)$
Move	$O(n)$	$O(n)$	$O(n^3)$

Feasible delete. Checking whether a solution is feasible after the node operator deletes a node u_j in T_i requires linear complexity with respect to the number of clients in S_j since it is necessary to check whether the new distance from the root to the furthest leaf for all nodes $u_k \in S_j$ satisfies the distance constraint:

$$f^i_{p_j} + b^i_k + d_{p_j,k} \leq \lambda$$

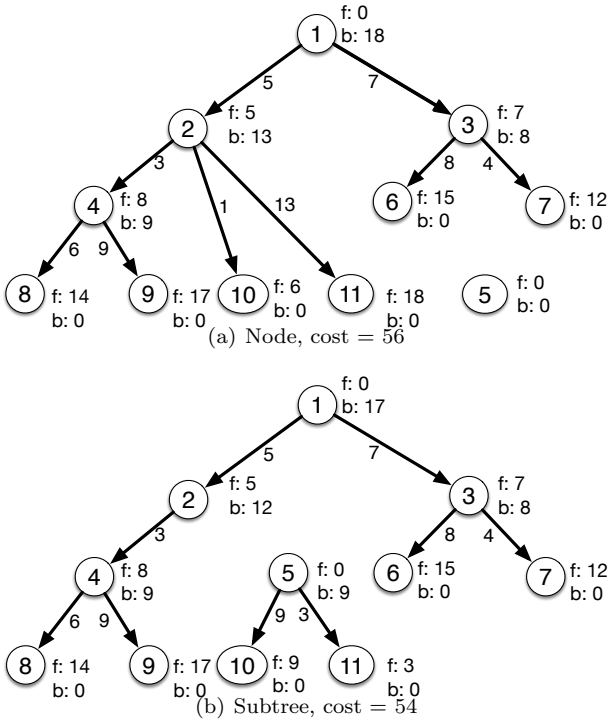
Delete. Deleting a node or the subtree emanating from the node u_j in T_i requires a linear complexity with respect to the number of clients of facility m_i. For both operators, it is necessary to update $b^i_{j'}$ for all the nodes j' in the path from the facility m_i to the client u_{p_j} in T_i. In addition, the node operator updates $f^i_{j'}$ for all the nodes j' in the subtree emanating from u_j. After deleting a node u_j or a subtree emanating from u_j, the objective function is updated as follows:

$$obj := obj - c_{j,p_j}$$

Furthermore, the node operator needs to add to the objective function the cost of disconnecting each successor element of u_j and reconnecting them to u_{p_j}.

$$obj := obj + \sum_{k \in S_j} (c_{k,p_j} - c_{kj})$$

Figure 4 shows an example of deleting a node and a subtree in the tree. In this example, deleting node 5 (Figure 4(a)) involves updating the immediate predecessor of node 10 and 11. The backward distance (i.e., b) of nodes 1 and 2



(a) Node, cost = 56

(b) Subtree, cost = 54

Fig. 4 Resulting trees after deleting node 5 using: (a) node move operator and (b) subtree move operator. Edges denote the cost and distance of connecting two nodes.

needs to be updated due to the addition of edge $\langle 2, 11\rangle$. Notice that this edge introduces a higher backward distance to a leaf-node, i.e., 13, which is greater than the distance to the other two successors (i.e., 12 for node 4 and 1 for node 10). Additionally, we update the cost accordingly since we are removing edges $\langle 2, 5\rangle$, $\langle 5, 10\rangle$, $\langle 5, 11\rangle$, and adding edges $\langle 2, 10\rangle$ and $\langle 2, 11\rangle$. The final cost after deleting node 5 is 56.

Deleting the subtree emanating from node 5 involves updating the forward distance (i.e., f) of all nodes in the emanating subtree, and updating the backward distance from all nodes in the path from the predecessor of node 5 (i.e., node 2) to node 1 (root-node) as shown in Figure 4(b). The cost is re-adjusted after the deletion of edge $\langle 2, 5\rangle$. The cost after the deletion of the edge is 54.

Feasible insert. Checking feasibility for a move can be performed in constant time by using f_j^i and b_j^i. If u_j is inserted between the two nodes of edge $\langle u_p, u_q\rangle$ then we check the following:

$$f_p^i + d_{pj} + d_{jq} + b_q^i \leq \lambda$$

If u_j is inserted as a new successor of u_p we check the distance constraint as follows:

$$f_p^i + d_{pj} + b_j^i \leq \lambda$$

To illustrate the feasibility check we recall that in Figure 4 we removed node 5 from the solution. Now let us assume we want to check whether breaking the existing edge $\langle 1, 3\rangle$ is a valid move or not. To verify the distance constraint the algorithm checks that the new distance from the root to the farthest leaf in the tree satisfies the length constraint (i.e., new distance is less than or equal to λ). Therefore, we check that the forward distance of the new potential predecessor node 1 to node 5 (added node), plus the distance of the new edge $\langle 5, 3\rangle$, plus the backward distance of node 3 is less than or equal to λ.

Now let us consider the case of the tree operator where we break the existing edge $\langle 1, 3\rangle$ in Figure 4(b). Similarly to what is done with the node operator, we check the feasibility of the new potential location by computing the forward distance of the new potential predecessor (i.e., node 1), plus the backward distance of the root node of the implicated subtree (i.e., node 5).

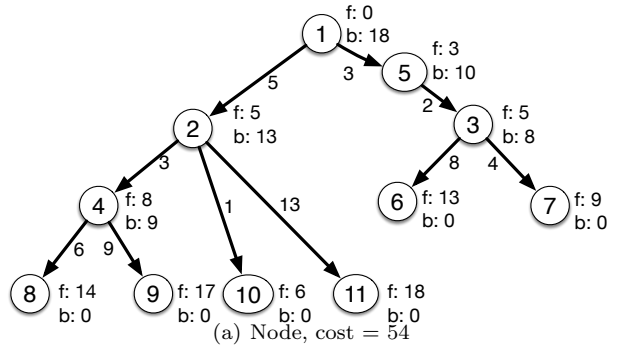
Best location. Selecting the best location involves traversing all clients associated with the facility and selecting the one with the maximum reduction in the objective function. Both operators need to traverse the tree in order to evaluate the cost of breaking all existing edges or adding a new node or subtree in the current solution. In this paper we use a depth-first exploration of the tree.

Insert. A move can be performed in linear time. We recall that this move operator might replace an existing edge $\langle u_p, u_q\rangle$ with two new edges $\langle u_p, u_j\rangle$ and $\langle u_j, u_q\rangle$. This operation requires to update f_j^i for all nodes in the emanating tree of u_j, and b_j^i in all nodes in the path from the facility acting as a root node down to the new location of u_j. The objective function must be updated as follows:

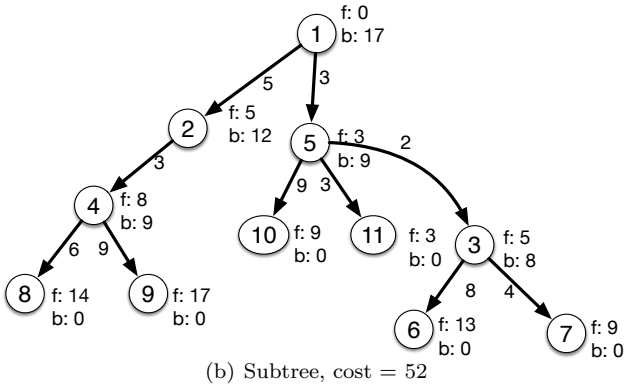
$$obj := obj + c_{pj} + c_{jq} - c_{pq}$$

Following our previous example, we use Figure 6 to insert a node and a subtree in the current solution. In the case of the node operator (Figure 5(a)) it is necessary to update the forward distance of node 5 and all nodes emanating from this node, i.e., nodes 3, 6, and 7. This modification also requires a modification in the backward distance of all nodes in the path from node 5 to the node 1 (root node). The backward distance for node 1 is not affected by the new edge in the tree since all the leaf nodes originating from node 5 are closer than the current furthest node. The cost after inserting node 5 is 54.

The subtree operator (Figure 5(b)) also updates the forward distance of node 5 and all the nodes emanating from this node, i.e., nodes 10, 11, 3, 6, and 7. Furthermore, the operator also updates the backward distance of all the nodes in the path from the root-node to the node 5. Similarly to the case of the node operator, we update the cost with the two new edges. The cost obtained after inserting the subtree is 52.



(a) Node, cost = 54



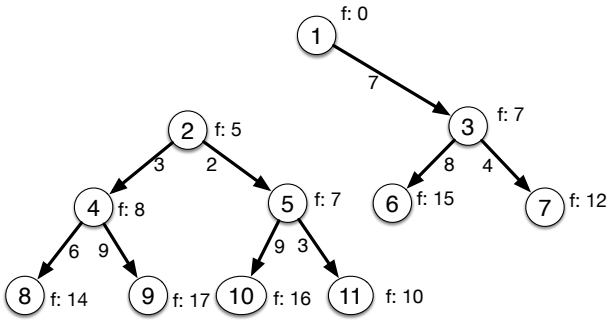
(b) Subtree, cost = 52

Fig. 5 Examples of inserting the node 5 using (a) node move operator and (b) subtree move operator. Edges denote the cost and distance of connecting two nodes.

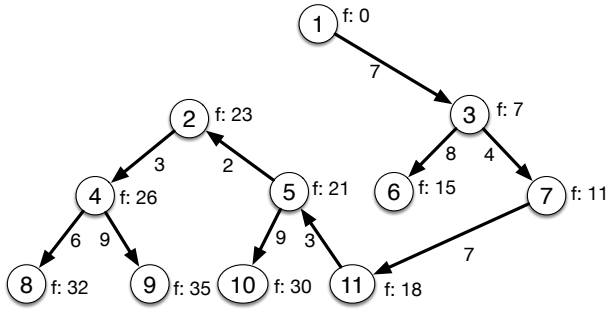
Now we switch our attention to the edge operator. This operator does not benefit from using b^i_j. The reason is that moving a given edge from one location to another might require changing the direction of a certain number of edges in the tree as shown in Figure 2(d). Deleting an edge requires constant complexity. This operation generates two separated subtrees and no data structures need to be updated. Checking the feasibility of adding an edge $\langle u_{p'}, u_{q'} \rangle$ to connect the two subtrees requires linear complexity. It is necessary to traverse the new tree to obtain the distance from $u_{q'}$ to the farthest leaf in the tree associated with facility i. Performing a move requires a linear complexity. It involves updating f^i_j for the new emanating tree of $u_{q'}$. The best location requires a cubic time complexity as the number of possible locations is bounded by n^2 (total number of possible edges for connecting the two subtrees) and for each possible move it is necessary to check feasibility. Due to the high complexity ($O(n^3)$) of the edge operator to complete a move, hereafter we limit our attention to the node and subtree operators. Nevertheless, we use Figure 6 to illustrate how to delete edge $\langle 1, 2 \rangle$ and insert a new edge $\langle 7, 11 \rangle$ in the solution.

As pointed out earlier in this paper, maintaining the backward distance is not providing any reduction in the complexity of the moves for the edge operator. Notice that deleting an edge does not necessarily requires to update the distance of the affected nodes in the tree. In this case, only the affected edge is removed from the solution.

The edge operator updates the forward cost of the tree emanating from the end node of the new edge $\langle 7, 11\rangle$ (i.e., node 11). Notice that the direction of some edges must be also changed (i.e., $\langle 2, 5\rangle$ and $\langle 5, 11\rangle$). As pointed out checking the feasibility of a move using the edge operator requires to traverse the new subtree emanating from the end-node of the new edge to compute the forward cost from the root-node to the furthest leaf in the tree.



(a) Delete, cost = 51



(b) Move, cost = 58

Fig. 6 Resulting trees after deleting and inserting using the edge move operator

Disjointness. To ensure disjointness among spanning-trees we maintain a $2{\cdot}|\mathcal{U}|$ Matrix, where $|\mathcal{U}|$ represents the number of clients. Checking disjointness for every client requires constant time complexity and only involves checking that the two integers indicating the predecessors in the primary and secondary facilities are different. Therefore the complexities presented in Table 1 remain valid when disjointness is considered.

Move. As pointed out in Table 1 completing a move for the node and subtree operators requires linear time complexity and it involves deleting a node u_j (or the complete subtree emanating from u_j), checking feasibility for all potential candidates locations, and then inserting the node or subtree in the most suitable location. We randomly select u_j in order to balance the greediness and the complexity of the move. Alternatively, one can select the best deletion-insertion pair. However, this option make the complexity of completing a move $O(n^2)$ as it involves deleting all nodes in the current solution and re-inserting them. Additionally, the use of the best deletion-insertion pair may lead to premature convergence to local minima.

In [23] the authors proposed two move operators for the RDCMST. *Edge-Replace* is similar to the edge operator described before. The difference is that in this case the authors only take into consideration the distance constraint, and therefore, the authors use the cheapest alternative edge to reconnect the two trees. Similarly to the edge operator *Edge-Replace* might change the direction of some affected edges and therefore the new solution might not satisfy the edge disjoint constraint. An alternative solution might be to limit the set of candidate neighbours to those not changing the direction of the tree. This would be a particular case of the subtree operator where the subtree is reconnected to an existing leaf of the current solution. *Component-Renew* removes an edge in the tree. Nodes that are separated from the root node are sequentially added in the tree using Prim's algorithm. Nodes that violate the length constraint are added in the tree using a pre-computed route from the root node to any node in the tree. It is worth noticing that the *Component-Renew* operator cannot be applied to ERDCMST as the pre-computed route from the root node to a given node might not be available due to the disjoint constraint.

5 Sequential Algorithm

In this section we describe a constraint-based local search algorithm, which is parameterised by a move operator.

A (node or subtree) move is composed of four sub-operations: *delete*, *best location*, *insert*, and *feasibility check*. A move involves removing a node or a subtree from the tree and adding it in the best location. In order to find the best location a sequence of feasibility checks are carried out.

The location of a node in a tree is defined by its parent node and its set of successor nodes. If the node-operator is used then a new location for a node u_j can be found by either (1) selecting an existing edge $\langle u_a, u_b \rangle$ and inserting u_j between the two nodes such that the parent node of u_j is u_a and the set of successors is the singleton set $\{u_b\}$ or (2) selecting any node u_a and adding a new edge $\langle u_a, u_j \rangle$ such that the parent node of u_j is u_a and the set of successor nodes is empty. Similarly, if the subtree-operator is used then a new location for a subtree emanating from a node u_j can be found by either (1) selecting an existing edge $\langle u_a, u_b \rangle$ and inserting u_j in between the two nodes such that the parent node of u_j is u_a and the set of successors of u_j is updated by adding

the node u_b or (2) selecting a node u_a and adding a new edge $\langle u_a, u_j \rangle$ such that the parent node of u_j is u_a and the set of successor nodes of u_j remains same. We use $Locations(u_j, T_i, \textit{move-op})$ to denote all the possible locations of a node (or a subtree emanating from) u_j in tree T_i using *move-op*.

Let *list* be a set of potential nodes for which we might be able to find better locations. If the *list* is empty then it means that the algorithm reaches to a local minima for a chosen move operator. We also compute a graph which we call *facility connectivity graph*, denoted by fcg. The vertices in the graph represent the facilities and an edge between a pair of facilities represent that the facilities share at least 2 clients. Therefore, we can observe that if there is an edge $\langle u_i, u_j \rangle$, then a change in the tree of a facility u_i might help in finding better locations for some clients in the tree associated with u_j. Notice that if two facilities do not share at least 2 clients then they are independent from disjointness point of view. Figure 7 shows an example of a *fcg* of a problem with 4 facilities and 11 clients: clients $\{u_1, u_2, u_3, u_4\}$ are common to m_1 and m_2; clients $\{u_7, u_8, u_9\}$ are common to m_2 and m_4; clients $\{u_5, u_6\}$ are common m_1 and m_4; and clients $\{u_{10}, u_{11}\}$ are common to m_1 and m_3.

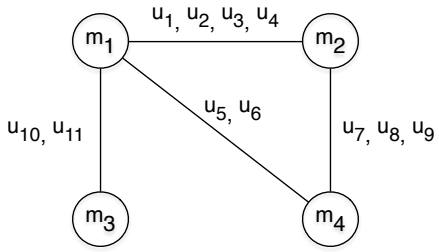


Fig. 7 Example of the facility connectivity graph (*fcg*) for a problem with 4 facilities (i.e., m_1, m_2, m_3, and m_4), and 11 clients

The pseudo-code of our constraint-based local search is showed in Algorithm 2. It starts by initialising *list* and *fcg* (Lines 2 and 3). It repeatedly selects a client u_j of a facility m_i randomly from Tree T_i (line 4), save its current location and deletes it from the tree (Lines 6-7). The delete operation depends on the move operation (*move-op*) and deletes a single node or the entire subtree emanating from the node. The cost of the current location is saved in *cost* (Line 8). For a chosen node or subtree, in each iteration (Lines 9-19), the algorithm identifies the best location. Line 10 verifies that the new move is not breaking any constraint and *CostLoc* returns the cost of using a new location using a given move operator (Line 10). If the cost of the new location is better then the best set of candidates is reinitialised to that location (Lines 12-14). If the cost is equal to the best known cost so far then the best set of candidates is updated by adding that location (Lines 15-16). The new location for a given node is randomly selected from the best candidates if there are more than one (Lines 21).

Algorithm 2 ConstraintBasedLocalSearch (*move-op*, $\{T_1, \ldots, T_{|\mathcal{M}|}\}$)

1: $list := \{(m_i, u_j) | m_i \in \mathcal{M} \wedge u_j \in U_i\}$
2: $fcg := \{(m_i, m_j) | N_i \cap N_j| \geq 2\}$
3: **while** $list \neq \emptyset$ **do**
4: Select (m_i, u_j) randomly from *list*
5: **if** FeasibleDelete(T_i, u_j, *move-op*) **then**
6: $oldLoc := (u_{p_j}, S_j)$
7: Delete(T_i, u_j, *move-op*)
8: $cost :=$ CostLoc((u_{p_j}, S_j), u_j, *move-op*)
9: **for** (u'_{p_j}, S'_j) in $Locations(u_j, T_i, move\text{-}op) - \{oldLoc\}$ **do**
10: **if** FeasibleInsert($(u'_{p_j}, S'_j), u_j, move\text{-}op$) **then**
11: $cost' :=$ CostLoc((u'_{p_j}, S'_j), u_j, *move-op*)
12: **if** $cost' < cost$ **then**
13: $BestLoc := \{(u'_{p_j}, S'_j)\}$
14: $cost := cost'$
15: **else if** $cost' = cost$ **then**
16: $BestLoc := BestLoc \cup \{(u'_{p_j}, S'_j)\}$
17: **end if**
18: **end if**
19: **end for**
20: **if** $BestLoc \neq \emptyset$ **then**
21: Select (u'_{p_j}, S'_j) randomly from *BestLoc*
22: $loc := (u'_{p_j}, S'_j)$
23: $list := list \cup \{(m_k, u_l) | (m_i, m_k) \in fcg \wedge u_l \in N_k \cap (\{u_j\} \cup (S_j \cup S'_j))\}$
24: $list := list \cup \{(m_i, u_l) | u_l \in N_i\}$
25: **else**
26: $loc := oldLoc$
27: **end if**
28: $T_i :=$Insert(T_i, *loc*, u_j, *move-op*)
29: **end if**
30: $list := list - \{(m_i, u_j)\}$
31: **end while**
32: return $\{T_1 \ldots T_n\}$

Instead of verifying that a local minima is reached by exhaustively checking all moves for all clients of all facilities, we maintain a list of pairs of clients and facilities. The list is initialised in Line 1 with all pairs of facilities and clients. In each iteration a pair consisting of a facility and a client, (m_i, u_j) is selected and removed from the list (Line 30). When *list* is empty the algorithm reaches a local minima. If an improvement is observed then one simple way to ensure the correctness of local minima is to populate *list* with all the pairs of facilities and clients. However, this can be very expensive in terms of time. We instead exploit the *facility connectivity graph* (*fcg*) where an edge between two facilities is added (in Line 2) if they share at least two clients. If an edge between m_i and m_k exists in *fcg* then it means that if there is a change in T_i then it might be possible to make another change in T_k such that the total cost can be improved. Consequently, we only need to consider the clients of affected facilities. This mechanism helps in reducing the time significantly by reducing the number of useless moves.

We say that a client in a tree is affected by a move if new opportunities for reducing the cost by repositioning the client in the tree become available after the move. We consider two cases: (a) the opportunities that are created in another tree T_k by a move in T_i (which correspond to the clients added in Line 23), and (b) the opportunities that are created within T_i itself (which correspond to the clients added in Line 24).

As a move takes place within a single tree, a client in T_k is only affected if its predecessor changes. Notice that u_j will alway belong to the set of affected clients in T_k since the move is changing its predecessor regardless of where the client is repositioned. If we are moving a node, the additional set of affected clients corresponds to $(S'_j \cup S_j)$. If we are moving a subtree, then the set of affected clients depends on whether we are breaking an existing edge $\langle u_p, u_q \rangle$ when repositioning the subtree or using a leaf of the tree as a new predecessor of u_j. In the former case the set of additional clients affected is $\{u_q\}$. In the latter there are not additional clients affected. In the implementation we just use $u_j \cup (S'_j \cup S_j)$, which is a superset in all cases.

The set of affected clients in T_i is more complex to characterise since the move is not only relaxing the disjointness constraint but also the length constraint. Indeed clients that were not feasible descendents of u_j might become feasible after repositioning u_j. Our current implementation is adding $T_i \setminus \{u_i\}$ to the list of affected clients, which is a superset of the actual set of affected clients in T_i.

To illustrate the behaviour of the algorithm let us revisit the example of Figure 2(b), where the list of affected clients for the node operator are the affected node (i.e., Node 5), the list of successors before the move (i.e., Nodes 10 and 11), and the list of successors after the move (i.e., Node 3). Alternatively, for the subtree operator, the affected clients are the elements in $S'_j \setminus S_j$. This set is empty unless the insertion of the subtree is breaking an existing edge $\langle u_p, u_q \rangle$, in which case the only affected node is u_q. In the example of Figure 2(c), the list will be populated with the affected node (i.e., Node 5) and Node 3.

The perturbation phase of the algorithm works similar to Algorithm 2, by selecting a random node, deleting a node or a subtree in the current solution, and then instead of selecting the best location, the algorithm selects a random valid location. In particular, for the perturbation phase, we replace Lines 10-18 with Line 16, i.e., $\mathit{BestLoc} := \mathit{BestLoc} \cup \{(u'_{p_j}, S'_j)\}$. Therefore, the algorithm selects a new random location for u_j.

We remark that Algorithm 2 can tackle both RDCMST and ERDCMST. In the case of RDCMST, there would be only one facility, and *Feasible* would only check the path-length constraint.

6 Parallel Algorithm

Parallelisation has been widely studied to speed-up and improve performance of local search algorithms to tackle a large variety of problems including:

TSP [8], Capacitated Network Design [10], Steiner Tree [29], SAT [17], and CSPs [9] just to name a few. These approaches employ the Multi-walk and/or Single-walk framework [28] to devise the parallel algorithm. In particular we focus our attention on constraint-based local search solvers.

6.1 Multi-walk and Single-walk

Multi-walk (also known as parallel portfolio) consists in executing several algorithms (or different copies of the same one with different random seeds) in parallel, with or without cooperation, until a solution is found or a given timeout is reached. The implicit assumption is that different processes would handle different parts of the search space. The multi-walk method has two important properties. First, no load balancing is required to parallelise the sequential algorithm. Second, the speedup of the parallel algorithm strictly depends on the performance of the sequential one. As discussed in the literature (e.g., [26] and [25]) a high variance of the sequential algorithm usually means good parallel speedup factors.

Single-walk methods consist in using parallelism inside a single search process. In this approach, a typical way to develop the algorithm consists in parallelising the exploration of the neighbourhood, e.g., dividing the neighbourhood into several sub-neighbourhoods and searching them in parallel to find the best move.

We observe the two mentioned levels of parallelism in the context of SAT (see [3] for a recent survey). An elaboration on multi-walk approaches with and without cooperation can be found in [1, 4], where the cooperation is implemented by sharing the best solution in order to properly craft a new starting point. In SAT, the single-walk approaches are implemented by flipping multiple variables at the same time [21].

The Comet solver [18] has been proposed in the context of constraint-based local search. Comet provides abstractions for implementing multi-walk parallelism (with and without cooperation) and single-walk parallelism.

6.2 Parallel Moves for ERDCMST

The aim of our parallel approach is to reduce the elapsed time for finding a target solution (i.e., optimal or near-optimal solution) by taking advantage of the availability of multiple cores and processors. In order to accomplish this, we propose a novel approach to perform multiple moves in parallel, which can be applied both in single-walk and multi-walk settings.

A move for ERDCMST can be defined as selecting and removing a node from a tree and adding it back to the tree preferably to a different location in the tree. The general idea is to partition the set of all moves in such a way that when multiple moves are performed by selecting them from different elements of the partition no constraint is violated. We use this approach to develop a parallel algorithm for the ERDCMST problem.

Algorithm 3 Random Independent set(*fcg*, *card*)

```
1: S := ∅
2: while fcg ≠ ∅ and |S| < card do
3:     v := random vertex in fcg
4:     S := S ∪ v
5:     Remove v and its neighbours from fcg
6: end while
7: return S
```

Informally speaking the algorithm takes into account the disjoint constraint to divide the problem space into mutually exclusive subproblems and asynchronously perform multiple moves in parallel. We use the *fcg* to decompose the problem. Let us recall that if N facilities are not sharing any client we can use the LS algorithm to optimise the local solution of individual facilities in parallel. For instance, in Figure 7 m_3 and m_2 are not sharing clients, so we can improve the solution by executing two parallel copies of Algorithm 2 limiting the input solution to m_3 for one core, and m_2 for the other one. In this section we describe two methods to decompose the problem. The first method computes a set of independent facilities. The second method randomly selects a set of facilities and resolves the conflicts between clients before executing the LS algorithm.

Let Γ_{ij} be the set of all potential predecessors of client u_j in T_i when considering either a node move or a sub-tree move. Ideally, we would like to find a set of nodes (or clients) whose sets of locations are pair-wise mutually exclusive so that moving all those nodes simultaneously in their trees is conflict-free. The advantage is that finding a best location for all such nodes can be done in parallel without restricting the access to the data-structures or creating duplicate copies of the same data-structure.

As the sets of locations for the selected nodes must be independent, we select at most one node from one tree. It is indeed possible to find two nodes in the same tree whose sets of potential predecessors are mutually exclusive, but finding such nodes is not straightforward. Therefore, the number of moves that can be performed simultaneously is bounded by the number of facilities.

As mentioned before, changing the location of a node within a tree T_i is not only constrained by the other nodes of T_i but also by the nodes of the other trees sharing nodes with T_i because of the disjoint constraint. In order to determine the number of subproblems, we use the previously defined facility connectivity graph. In particular, we explore the following two mechanisms:

1. **Independent set** defines partitions by computing independent sets in *fcg*. In this approach, as we know beforehand that the potential locations for the clients are conflict-free (w.r.t. disjointness), all elements in the independent set are safely executed in parallel without violating the disjoint constraint. Algorithm 3 computes a random set of independent elements in *fcg*. These elements will be then used in the parallel section of the algorithm to solve the problem. *card* refers to the cardinality of the independent set to be computed. Ideally, *card* should match the number of cores available. How-

ever, the degree of parallelism of the algorithm is bounded by the carnality of the maximum independent set in *fcg*. For instance, let us revisit the *fcg* in Figure 7. In this case, we will have at most 2 independent facilities (i.e., m_3 and m_4). Therefore, even if *card* is greater than 2, Algorithm 3 will return at most two partitions. In practice we expect sparse graphs in real networks, so the cardinality of independent sets should be more than a few tens of elements for real size networks.

2. **Random conflict** selects, uniformly at random, n facilities and resolves the conflicts between clients apriori. It is recalled that two facilities can be in conflict if and only if they share at least two clients. Let us say that two facilities m_i and m'_i are selected, and the clients u_j and $u_{j'}$ are connected to both facilities. Let $C = \Gamma_{ij} \cap \Gamma_{i'j'}$ be a non-empty set. To resolve the conflict we modify the sets Γ_{ij} and $\Gamma_{i'j'}$ such that they become mutually exclusive. We recall that Γ_{ij} ($\Gamma_{i'j'}$) represents the set of potential predecessors for u_j ($u_{j'}$) in the tree T_i (T'_i).
   - If $u_k \in C$ is already a predecessor of u_j in T_i then we remove u_k from $\Gamma_{i'j'}$, or vice-versa.
   - If $u_k \in C$ is a predecessor of neither u_j in T_i nor $u_{j'}$ in T'_i then we remove u_k randomly from either Γ_{ij} or $\Gamma_{i'j'}$. We can say that the algorithms decides beforehand to which set u_k should belong.

   Unlike *independent set* where the degree of parallelism is limited by the size of the maximum independent set, *random conflict* allows as many processes as the number of facilities in the problem, which in practice goes up to few hundreds of cores.

   The solution of a problem whose *fcg* is as the one in Figure 7 will start by randomly selecting a set of facilities, e.g., m_1, m_3, and m_4, and then the *random conflict* method resolves conflicts beforehand for conflicting clients. For instance, if edge $\langle u_{10}, u_{11} \rangle$ is present neither in the current solution of m_1 nor in the current solution of m_3, the algorithm randomly decides whether to forbid the edge in m_1 or in m_3. The algorithm repeats this process for all pairs of conflicting clients.

Algorithm 4 shows the Iterated Constraint-based Parallel Local Search algorithm (ICPLS) proposed in this paper. We start with an initial solution (Line 1). As it is the case in the sequential algorithm, s^* denotes the current incumbent solution of the problem, s'^* denotes the solution after perturbing the incumbent solution, s'^* denotes the best solution obtained after executing the constraint-based local search framework.

CreatePartitions (Line 5) computes a set of facilities P using either *independent set* or *random conflict*. *Independent set* computes a set of independent facilities, i.e., the facilities in the computed set do not share nodes between them, thus allowing changing the structure of each tree independently without violating the disjoint constraint. *Random conflict* resolve conflicts (if any) of the locations of the clients of the the set of facilities P by restricting their locations. For each facility $p_i \in P$ the algorithm performs (Lines 6-14), in parallel, the sequential local search algorithm for a given amount of time t to

Algorithm 4 Iterated Constraint-based Parallel Local Search(*move-op*, t)

```
1: s* := Initial Solution
2: repeat
3:     {s*_1, ... s*_n} := s*
4:     {s'_1, ... s'_n} := s*
5:     P := CreatePartition(s*)
6:     for each p_i ∈ P do in parallel
7:         while local time limit t for parallelism has not been reached do
8:             if s*_i is internally in a local minima then
9:                 s'_i := Perturbation(s*_i)
10:            end if
11:            s'*_i := ConstraintBasedLocalSearch(move-op, s'_i)
12:            s*_i := AcceptanceCriterion(s*_i, s'*_i)
13:        end while
14:    end parfor
15:    s* := {s*_1, ... s*_n}
16: until A given stopping criterion is met
```

explore the search space. As it is the case in the sequential algorithm, s^* denotes the current incumbent solution of the problem, s'^* denotes the solution after perturbing the incumbent solution, s'^* denotes the best solution obtained after executing the constraint-based local search framework. s_i^* (respectively $s_i'^*$ and s_i') denotes the solution (or tree) associated with partition p_i.

In the parallel section of the algorithm we differentiate between the local minima for each facility and the local minima of the problem. In the sequential algorithm we diversify the current incumbent solution after finding a local minima of the problem, i.e., a state in which no neighbour solution leads to an improvement in the objective. In the parallel algorithm we diversify (Lines 8-10) when the solution s_i associated with facility p_i is in internally in a local minima, but the global state of the whole problem is still unknown. Then, the Constraint-based Local Search procedure is invoked (Line 11) with the current solution s_i associated with p_i using the acceptance criterion of the sequential algorithm. We remark that the set of potential locations for the nodes at this point are mutually exclusive, and therefore Line 2 of the sequential algorithm is not needed.

The sequential algorithm scans the list of active clients (*list*) deleting clients from the list when they cannot improve the objective function. The perturbation starts when a local minima in the problem is reached (i.e., *list* = { }). Following the same approach in the parallel algorithm may introduce a considerable processor idle-time, in particular when approaching to a local minima. Therefore, we start the perturbation locally for each tree as soon as an internal local minima for a given tree is reached to minimise idle-time. Moreover, after applying a move in a tree, only nodes of the same tree are added to *list* to reduce synchronisation among processors (Line 22 in Algorithm 2).

7 Application: Long-Reach Passive Optical Networks

To demonstrate the effectiveness of our approach we consider a real-world problem arising in the domain of optical networks. Long Reach Passive Optical Networks (LR-PONs) are gaining increasing interest as they provide a low cost and economically viable solution for fibre-to-the-home network architectures [20]. An example of a Long-Reach PON is shown in Figure 8. In LR-PON the optical reach is about 100 km [11]. The longer reach allows to bypass Exchange sites, which eliminates electronic traffic processing in the metro nodes and the need for a dedicated metro network. The longer reach also reduces the number of active network nodes by as much as two orders-of-magnitude, while all electronic data processing can be removed from the exchanges, thereby reducing both cost and energy consumption. In this paper, we focus our attention in the backhaul network for connecting exchange-sites to metro nodes. We use the minimum lambda value, which corresponds to the maximum distance from an exchange-site to the nearest metro node.

In this architecture each metro node is connected to tens of thousands of customers via tens of hundreds of exchange-sites. A major fault occurrence would be a complete failure of the metro node that terminates the LR-PON, which could affect tens of thousands of customers. The *dual homing* protection mechanism for LR-PON enables customers to be connected to two metro nodes via local exchange site, so that whenever a single node fails all customers are still connected to a back-up [15]. Simply connecting two metro nodes to an exchange site is not sufficient to guarantee the connectivity because if a link is common in the routes of the fibre going from the exchange site to its two metro nodes then both metro nodes would be disconnected. The part of the LR-PON that one wants to protect is between the metro node and the old local exchange site. An important property of such a network is resilience to a single metro node failure. Nevertheless, connecting an exchange site to an additional metro node has a cost overhead. Here a metro node is a facility and an exchange-site is a client.

In the optical network fibres are distributed from the metro nodes to the exchanges through cables that form a tree distribution network. As the association between metro nodes and exchange sites is already given, we could treat each one-to-many relation (i.e., tree) independently if disjointness were not an issue. However, the paths from the metro nodes to the exchange sites may share edges since a pair of exchange-sites may be associated with the same pair of metro nodes. Therefore, we need to make sure that the routes that we choose for connecting the exchange sites to its metro nodes (i.e., main metro node and back up metro node) are disjoint. Otherwise, this would void the purpose of having double coverage. In Figure 9 we show two ways of connecting a given set of exchange sites to a metro node. In the first case (Figure9(a)) we are simply connecting each exchange site. Certainly the option of connecting each exchange site directly to the metro node leads to shorter connection paths. However, the drawback of connecting each exchange site directly is the total amount of cable used. In the second case (Figure 9(b)) we are computing

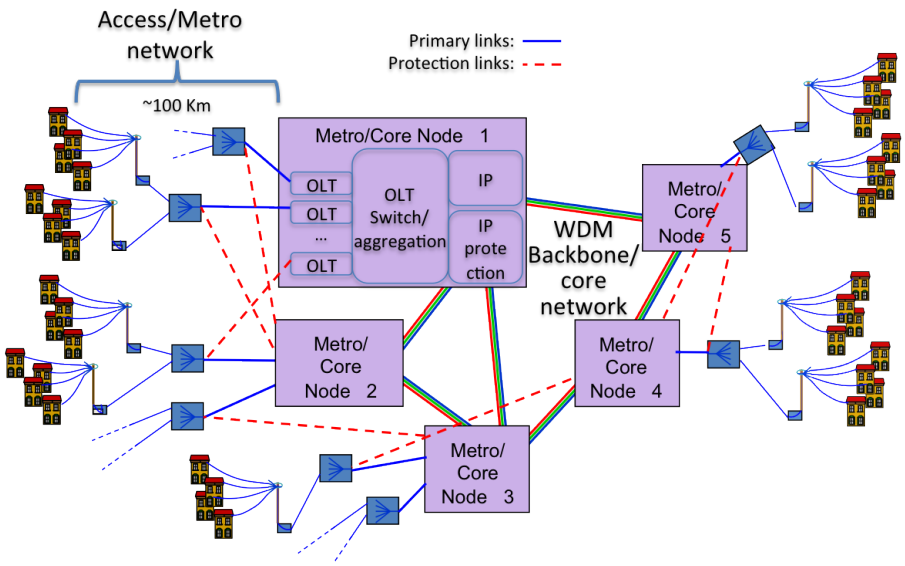
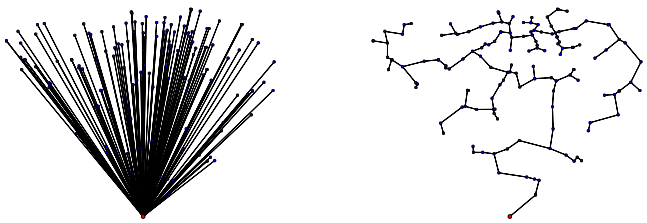


Fig. 8 Example of a Long-Reach Passive Optical Network

a minimum spanning tree rooted at the metro node. Certainly this option minimises the total length of cable but the drawback is that we might be violating the maximum cable distance allowed between the metro node and any of its exchange-site.



(a) (b)

Fig. 9 (a) Local exchanges are directly connected to the metro node. (b) Local exchanges are connected to the metro node through a spanning tree.

We are interested in both restricting the length of the paths and the total amount of cable used. Keeping both requirements is known to be a hard problem [13]. As mentioned before this problem is computationally complex [19]. Notice that our problem is even more complicated than the bounded spanning tree problem. The objective of our problem is to determine the optimal routes of cables in the context of an already existing association of metro nodes with

exchange sites such that the total cable length required for connecting each exchange site to two metro nodes is minimised subject to the maximum distance constraint and disjointness constraint. In other words, the idea is find two edge disjoint paths for all exchange-sites to their respective metro nodes by maximising sharing (since we want to minimise the cost of digging, for example), and reducing the amount of cable. Figure 10 shows a LR-PON solution example for the national education and research network (HEAnet) of Ireland with 14 MNs and 135 ESs.[2]

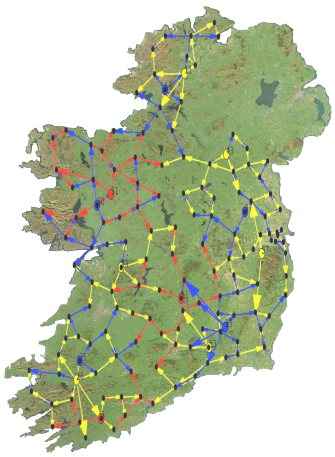


Fig. 10 Example of a LR-PON for the national education and research network (HEAnet) of Ireland with 14 MNs and 135 ESs where each ES is connected to two MNs through disjoint paths. In the plot the subnetwork of each metro node is associated with a colour. Two subnetworks may have the same colour if they do not share nodes (please see in colour).

8 Empirical Evaluation

In this section we present experimental results for the sequential and parallel version of the local search algorithm proposed in this paper to tackle ERDCMST. Moreover, we also demonstrate the effectiveness of the algorithm when compared to a dedicated algorithms for the RDCMST. We present results for random and real-world instances. The real-world instances correspond to the optical networks of three EU countries: Ireland, with 1121 exchange sites and 18, 20, 22, and 24 Metro Nodes; the UK, with 5393 exchange sites and 75, 80, 85, and 90 Metro Nodes; and Italy, with 10709 exchange sites and 140, 150, 160, and 170 Metro Nodes. Hereafter, we present experimental results for the ERDCMST problem with the sequential and parallel LS algorithms.

[2] The HEAnet network is fully described in http://www.heanet.ie/

All the experiments were performed in a 4-node cluster, each node features 2 Intel Xeon E5-2640 processors at 2.5 Ghz, and 64 GB of RAM memory. Each processor has 6 cores for a total of 12 cores per node. The local search algorithm was implemented in C++ and used openMP to implement the parallel version using shared memory. In all the experiments we use the following parameters for Algorithm 1: p=5%, 20 random moves in the perturbation phase of the algorithm, and the algorithm is stopping after a given time limit is reached. Additionally, we use the trivial solution, i.e., direct connections from the MNs to the ESs, as an initial solution for the algorithm.

8.1 ERDCMST Results: Sequential LS

We evaluate the proposed algorithms using the following scenarios:

- *Real-life*: We consider real-life instances from our industrial partners with real networks in Ireland, the UK, and Italy.
- *Random*: We generated 10 random instances extracted from the previous real-life network in Ireland. Each instance is generated by using 18 facilities and for each facility we randomly selected $|U| \in \{100, 200, \ldots, 1000\}$ nodes. Instances were generated by iteratively selecting a random client from the Irish dataset and balancing the load of clients per metro node.

In all our experiments we use the minimum λ value, i.e., 67 for Ireland and 62.5 for Italy and the UK. We validated these λ limits with our industrial partners.

Table 2 reports results for the *Random* experimental scenario, which depict the median value across 11 independent executions of the node and subtree operators; the best solution obtained with CPLEX; the best solution obtained with CPLEX using the solution of the first execution of LS with the subtree operator as warm start (LS+CPLEX); and the best known LBs for each instance obtained with CPLEX using a larger time limit.[3] The time limit for each local search experiment was set to 30 minutes, and 4 hours for the CPLEX-based approaches.

In these experiments we observe that the subtree operator generally outperforms the node operator. We attribute this to the fact that moving a complete subtree helps to maintain the structure of the tree in a single iteration of the algorithm. The node operator might eventually reconstruct the structure, however, more iterations would be required. CPLEX-based approaches report the optimal solution for 100 and 200 clients, while the median execution of the local search approaches reported the optimal solution for 100 clients, and the subtree operator reached the optimal solution in 5 out of the 11 executions for 200 clients. After $|U|$=500 LS dominates the performance for a margin ranging between 1% ($|U|$=500) to 12% ($|U|$=900). LS+CPLEX was only able

[3] In this paper CPLEX corresponds to solving the MIP model with IBM ILOG CPLEX Optimisation Studio version 12.5.1.

Table 2 Results for the small-sized instances of ERDCMST problem where $|M| = 18$, λ=67, 30 minutes time limit for LS approaches, 4 hours for CPLEX, and 5 hours for LS+CPLEX

$\|E\|$	LS (Subtree)	LS (Node)	CPLEX	LS+CPLEX	LB
100	**4674**	**4674**	**4674**	**4674**	4674
200	6966	6988	**6962**	**6962**	6962
300	8419	8575	**8404**	**8404**	8152
400	9728	10008	9728	**9721**	9329
500	**11203**	11672	11318	**11203**	10298
600	**11885**	12559	12276	11924	10517
700	13148	13981	13812	**13140**	11485
800	14040	15133	15118	**13977**	12402
900	**14770**	16098	16438	14839	12860
1000	**15962**	17479	18174	16009	13943

Table 3 Results for Ireland, UK and Italy with 30 minutes time limit (wall time) for the LS algorithm and 4 hours time limit for CPLEX (wall time)

Country	$\|M\|$	Subtree	CPLEX	LB	Gap-Subtree	Gap-CPLEX
	18	**17155**	26787	14809	13.67	44.71
Ireland	20	**16884**	83746	14845	12.07	82.27
$\|U\|$=1121	22	**16715**	79919	14990	10.32	81.24
	24	**16173**	26918	14570	9.91	45.87
	75	**66367**	285014	54720	17.54	80.80
UK	80	**65380**	301190	54975	15.91	81.74
$\|U\|$=5393	85	**64189**	281546	55035	14.26	80.45
	90	**62763**	220041	55087	12.23	74.96
	140	**90796**	–	76457	15.79	–
Italy	150	**89519**	–	76479	14.56	–
$\|U\|$=10708	160	**89497**	–	76794	14.19	–
	170	**88497**	–	77013	12.97	–

to improve the average performance of the subtree operator (reached after executing LS for 1 hour) in a very small factor, i.e., up to 0.4% for $|U|$=800, after running CPLEX for 4 hours. We also experimented with instances with $|U|$<100 and $|U|$>800. In the first case the three algorithms and the mixed approach (LS+CPLEX) reported similar results. In the second case only LS with the subtree operator was able to provide good quality solutions with a Gap of 10% w.r.t. the LB.

Our second set of experiments for the sequential version of the constraint-based local search algorithms are showed in Table 3 where we report results for real ERDCMST instances from Ireland, Italy, and the UK. In this case, we used a time limit of 30 minutes for LS (using the subtree move operator), and four hours for CPLEX. As it can be observed, LS dominates the performance in all these experiments, and once again the solution quality of LS does not degrade with the problem size. Indeed, the gap with respect to the LB for local search varies from 9.9% to 13.6% for Ireland, 12.2% to 17.5% for UK, and 12.9% to 15.7% for Italy. CPLEX ran out of memory when solving instances from Italy. We report '–' when no valid solution was obtained. For the UK

instances CPLEX also ran out of memory before the time limit. Once again we would like to recall that algorithms such as BKRUS, PBH, and KBH cannot be used for the ERDCMST problem as they are dedicated algorithms for the RDCMST that rely on the use of shortest paths to build valid solutions. However, in the ERDCMST problem such path might not be available due to disjointness. Figure 11 depicts a boxplot for each country with the final cost of the experiments of the sequential algorithm. As it can be observed the algorithm reports a very small variance in cost of the solutions, and even the worst sequential execution is better than the CPLEX algorithm.

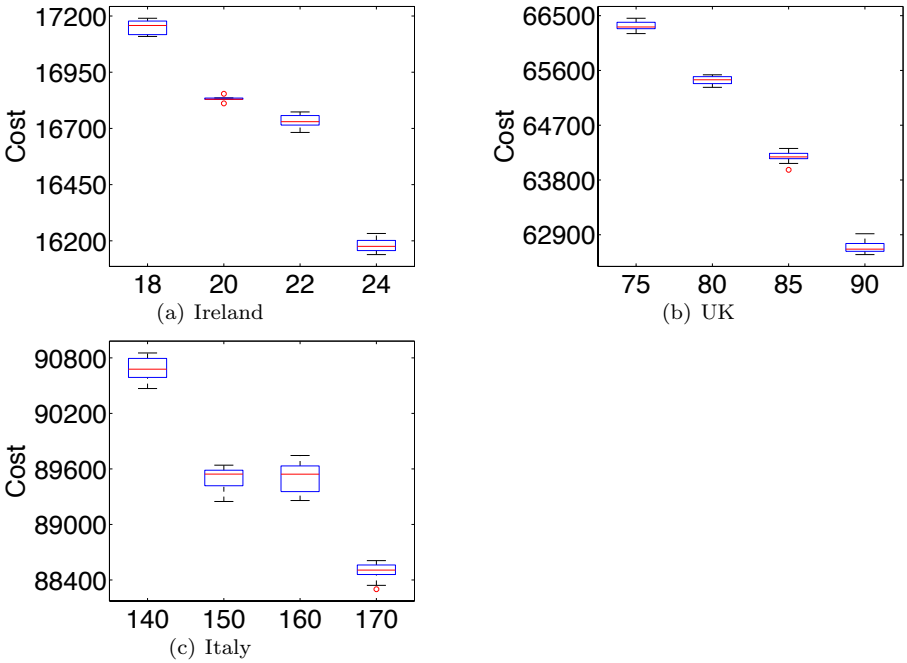


Fig. 11 Boxplot with the cost of the 11 executions of the sequential algorithm (using the subtree operator) and 30 minutes time limit for Ireland, UK, and Italy

Finally, we wanted to evaluate alternative heuristics to drive the local search algorithm. Table 4 shows a comparison of the cost and the total number of local minimum observed for the original subtree operator and a modified version subtree$^+$. subtree$^+$ selects the best deletion-insertion pair rather than deleting a random subtree from the solution and inserting it in the best location. The original subtree operator as depicted in Section 4 provides better solutions than using the best deletion-insertion pair. We observe on average 18% improvement for Ireland, 72% improvement for the UK, and 145% improvement for Italy. As pointed out earlier in Section 4 selecting a random node for deletion provides a good balance of greediness with randomness and selecting the best deletion-insertion pair (i.e., subtree$^+$) quickly converge to

Table 4 Results for Ireland, UK, and Italy with 30 minutes time limit (wall time) for the original subtree operator and subtree$^+$ a modified version selecting the best deletion-insertion pair in the intensification phase. Cost and #LM denotes the total cost and local minima observed during the search.

Country	$\|M\|$	Subtree		Subtree$^+$	
		Cost	#LM	Cost	#LM
Ireland $\|U\|$=1121	18	**17155**	**10694**	20736	51778
	20	**16884**	**11630**	20066	59839
	22	**16715**	**13353**	19525	60780
	24	**16173**	**14344**	18967	62097
UK $\|U\|$=5393	75	**66367**	**2184**	113705	37112
	80	**65380**	**2352**	111636	42748
	85	**64189**	**2650**	109649	43567
	90	**62763**	**2808**	110047	41332
Italy $\|U\|$=10708	140	**90796**	**1041**	222305	35910
	150	**89519**	**1117**	220673	36306
	160	**89497**	**1279**	217715	36807
	170	**88497**	**1424**	218056	37972

local minimum. This can be observed in the fact that subtree$^+$ reaches considerably more local minimum than the original subtree. For the UK and Italy the gap is of more than one order of magnitude.

8.2 ERDCMST Results: Parallel LS

In this section we evaluate the performance of the proposed parallel local search algorithm. To this end we use the same real-world instances used for the sequential algorithm for Ireland, the UK, and Italy. In this section we limit our attention to the subtree operator as it greatly outperforms the node operator in the sequential setting.

We define the *gain* of the parallel algorithm as the relative percentage gain w.r.t. the sequential algorithm in the cost solution after a given time limit and using given number of cores. Let $C(t, inst, c)$ be the cost of the best solution obtained after t seconds using c cores to solve $inst$. Let $T(cost, inst, c)$ be the time to reach a solution whose cost is at least as good as $cost$ for a given instance $inst$ using c cores.

$$Gain(inst, c, t) = \frac{C(t,inst,1)-C(t,inst,c)}{C(t,inst,c)} \times 100$$

Tables 5, 6, and 7 show the results of the empirical evaluation of the parallel algorithm. In these tables we present the cost of the solution for the sequential algorithm, the parallel algorithms, and the relative cost gain of the parallel algorithm after the 30-minute time limit. We use the multi-walk (MW) framework, i.e., executing multiple copies of the algorithm with different random seeds, as a baseline for comparison. We also include the proposed parallel algorithms using both *random conflict* (RC) and *independent set* (IS) for selecting multiple moves. For each instance and each approach we report the median value across 11 executions with a time-limit of 30 minutes.

Figure 12 shows the average performance evolution of the algorithms (parallel and sequential) to tackle one instance for each dataset, i.e., Ireland with 18 facilities, the UK with 75 facilities, and Italy with 140 facilities. The x-axis indicates the runtime and the y-axis the median quality of the solution of the 11 executions.

Table 5 Performance summary of the parallel algorithms (Multi-walk (MW), *random conflict* (RC) and *independent set* (IS)), with a 30-minute time limit (wall time).

Country	$\|M\|$	Seq	Alg	4 Cores		8 Cores		12 Cores	
				Cost	Gain	Cost	Gain	Cost	Gain
Ireland $\|U\|$=1121	18	17155	MW	17110	+0.26	17092	+0.37	17085	+0.41
			RC	17293	-0.83	17287	-0.86	17266	-0.77
			IS	17276	-0.69	17324	-0.76	17307	-0.85
	20	16884	MW	16841	+0.26	16829	+ 0.33	16828	+0.33
			RC	17001	-0.68	16998	-0.78	17015	-0.75
			IS	17007	-0.73	17014	-0.69	17006	-0.76
	22	16715	MW	16704	+0.07	16691	+0.14	16686	+0.17
			RC	16891	-1.01	16888	-1.05	16896	-1.04
			IS	16886	-1.00	16896	-1.02	16884	-0.98
	24	16173	MW	16152	+0.13	16148	+0.15	16136	+0.23
			RC	16315	-0.93	16318	-1.05	16347	-1.05
			IS	16327	-0.86	16327	-0.89	16323	-0.86

Table 6 Performance summary of the parallel algorithms (Multi-walk (MW), *random conflict* (RC) and *independent set* (IS)), with a 30-minute time limit (wall time).

Country	$\|M\|$	Seq	Alg	4 Cores		8 Cores		12 Cores	
				Cost	Gain	Cost	Gain	Cost	Gain
UK $\|U\|$=5393	75	66367	MW	66153	+0.32	66126	+0.36	66093	+0.41
			RC	65083	+1.99	64988	+1.73	64972	+1.82
			IS	65083	+1.96	64980	+1.81	64981	+1.89
	80	65380	MW	65328	+0.08	65282	+0.15	65265	+0.18
			RC	64290	+1.65	64227	+1.35	64217	+1.53
			IS	64328	+1.62	64207	+1.45	64195	+1.55
	85	64189	MW	64168	+0.03	64146	+0.07	64131	+0.09
			RC	63487	+1.11	63433	+0.97	63421	+1.02
			IS	63486	+1.10	63403	+1.01	63414	+1.04
	90	62763	MW	62726	+0.06	62651	+0.18	62641	+0.19
			RC	62210	+0.86	62171	+0.73	62153	+0.84
			IS	62260	+0.81	62191	+0.73	62140	+0.87

Ireland instance (Table 5 and Figure 12(a)). We remark that the local search algorithms finds a very good solution within a very short time window (GAP of up to 13% with respect to to the lower bound), and the variance between independent executions of the algorithm is very low. For this reason, when increasing the number of cores we observe very little difference in the perfor-

Table 7 Performance summary of the parallel algorithms (Multi-walk (MW), *random conflict* (RC) and *independent set* (IS)), with a 30-minute time limit (wall time).

Country	$\|M\|$	Seq	Alg	4 Cores		8 Cores		12 Cores	
				Cost	Gain	Cost	Gain	Cost	Gain
Italy $\|U\|$=10709	140	90796	MW	90669	+0.14	90633	+0.18	90621	+0.19
			RC	88573	+2.51	88382	+2.71	88332	+2.80
			IS	88529	+2.56	88358	+2.74	88330	+2.71
	150	89519	MW	89427	+0.1	89357	+0.18	89309	+0.24
			RC	87517	+2.27	87411	+2.42	87414	+2.41
			IS	87526	+2.26	87462	+2.37	87379	+2.43
	160	89537	MW	89421	+0.13	89360	+0.20	89309	+0.26
			RC	87679	+2.15	87579	+2.24	87525	+2.29
			IS	87666	+2.13	87564	+2.26	87528	+2.31
	170	88497	MW	88433	+0.07	88359	+0.16	88359	+0.16
			RC	86954	+1.76	86925	+1.83	86862	+1.89
			IS	86955	+1.77	86869	+1.86	86869	+1.88

mance of the algorithms.[4] We observe that the sequential algorithm is slightly better than the parallel ones with a percentage gain of between 0.67% to 1.05% within the 30-minute time limit. However, we would like to remark that the parallel algorithm reaches good solutions faster than the sequential one as depicted in Figure 12(a). This figure also helps to observe the difference between using *independent sets* and *random conflict* for computing the partitions. Notice that 8-core random partition reports a better performance than 12-core *independent set*. That is because the cardinality of the maximum independent set for this problem is 9 and the number of parallel processes is bounded by that number, thus voiding the advantage of having 3 more cores. *Random conflicts* allows as many parallel processes as the number of metro nodes in the problem.

UK instance (Table 6 and Figure 12(b)). The UK instance is about four times bigger (with respect to the number of clients) than the Irish instance. Except for the multi-walk approach (where no significance improvement is seen), we observe that the parallel algorithms improve the quality of the solutions when increasing the number of cores. Summing up, the observed performance gain ranges from 0.81% to 1.99% (4 cores), from 0.73% to 1.81% (8 cores), and 0.84% to 1.89% (12 cores). Moreover, as depicted in Figure 12(b), the parallel algorithm based on single walk also reaches a very high quality solution much faster than the sequential algorithm, and the performance increases as the number of cores increases. However, in this case, we observe similar performances between *independent set* and *random conflict*. That is because the independent sets are always larger than 12 and therefore both approaches exploit parallelism as much as possible.

Italy instance (Table 7 and Figure 12(c)). The largest performance improvement of the parallel algorithm is observed for Italy (Table 7). We attribute

[4] Similar behaviour for other local search algorithms has been observed in [2] in the context of the Satisfiability problem.

this to the size of the problem: the larger the problem the better the parallel algorithms perform. Here we observe a performance gain ranging from 1.76% to 2.59 (4 cores), 1.83% to 2.74% (8 cores), and 1.88% to 2.71% (12 cores). Similarly to the Irish and the UK datasets, Figure 12(c) shows the performance in time of the parallel algorithm, and once again it can be observed that the parallel version reaches very good solutions faster than the sequential algorithm.

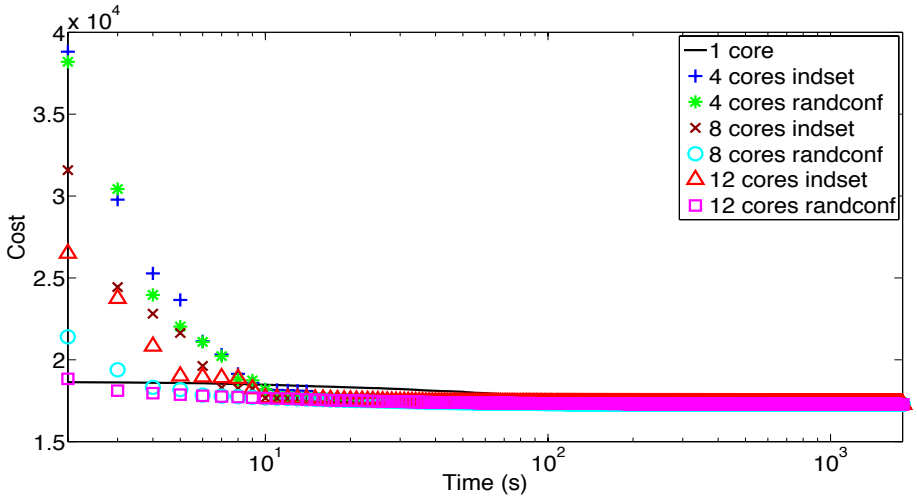
As pointed out before the sequential algorithm obtains better solutions for the Irish dataset after then 30 mins time limit, however, the parallel algorithm computes near-optimal solutions faster than the sequential algorithm. For instance, as showed in Tables 8, the average gain (with respect to to the sequential algorithm) after 100 seconds using 4 cores is 1.62% (independent set) and 1.69% (*random conflict*), and after 10 seconds we observe a gain of up to 4.29% for *random conflict* and 1.39 % for *independent set*. Once again we observe that when the cardinality of the maximum independent set is small with respect to the number of cores *random conflict* performs better than *independent set*. Interestingly, the relative gain of the parallel algorithm w.r.t. the sequential one is more than 100% in two occasions for the UK (up to 130% for RC with 12 cores and 100 secs) and in eight occasions for Italy (up to 182% for IS and RC with 12 cores and 100 secs).

Finally, Table 9 concludes the experiments reporting the average speedup factor (out of the four scenarios for each country) to reach the best solution after a given amount of time, i.e., 10, 100, 1000, 18000 seconds. We recall that the speedup factor is the gain in the speed of the parallel algorithm. We compute the speedup factor of the parallel algorithm with c cores after t seconds to solve a given instance $inst$ as follows:

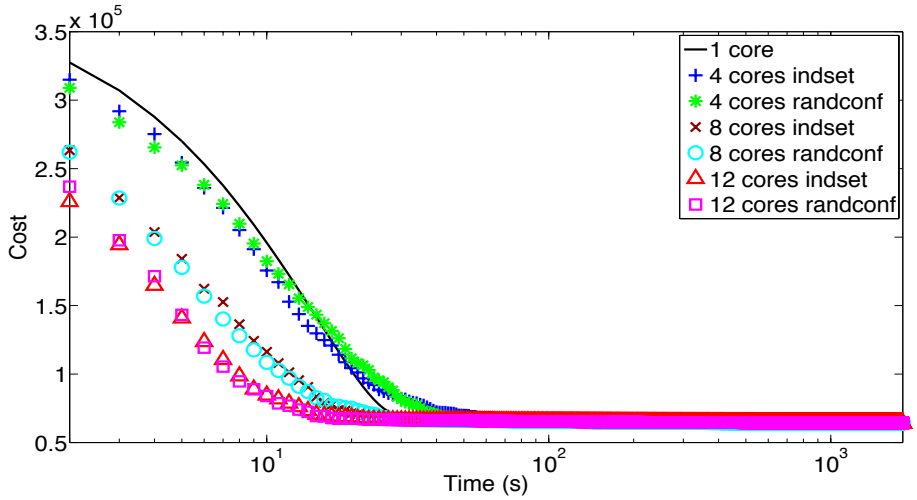
$$SpeedUp(inst, c, t) = \frac{T(C(t,inst,1),inst,1)}{T(C(t,inst,1),inst,c)}$$

Table 8 Average gain with different time settings for each country

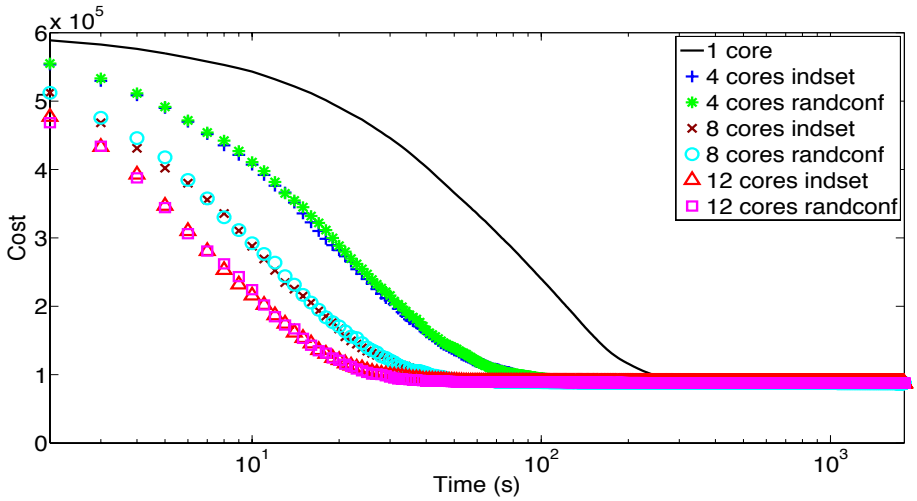
Country	Time	4 cores		8 cores		12 cores	
	Secs	IS	RC	IS	RC	IS	RC
Ireland	10	-3.70	-6.05	1.39	4.00	0.65	4.29
	100	1.64	1.56	1.62	1.69	1.66	1.64
	1000	-0.75	-0.79	-0.82	-0.78	-0.78	-0.82
	1800	-0.82	-0.86	-0.83	-0.93	-0.86	-0.90
UK	10	6.88	5.29	66.53	77.37	125.85	130.32
	100	3.86	3.82	4.70	4.58	5.06	5.01
	1000	2.13	2.15	2.37	2.34	2.42	2.40
	1800	1.37	1.40	1.25	1.19	1.33	1.30
Italy	10	30.99	30.34	78.36	79.13	131.27	128.50
	100	167.63	168.96	181.79	180.87	182.30	182.17
	1000	2.75	2.71	2.94	2.92	3.00	2.99
	1800	2.19	2.16	2.30	2.3	2.33	2.34



(a) Ireland $|M| = 18$



(b) UK $|M| = 75$



(c) Italy $|M| = 140$

Fig. 12 Solution cost vs. wall clock time (ranging from [2, 1800] seconds)

Table 9 Average speedup factor with different time settings for each country

Country	Time	4 cores		8 cores		12 cores	
	Secs	IS	RC	IS	RC	IS	RC
Ireland	10	0.86	0.79	1.19	1.76	1.06	2.62
	100	6.90	5.86	9.45	10.34	8.74	12.78
	1000	–	–	–	–	–	–
	1800	–	–	–	–	–	–
UK	10	1.07	1.02	2.13	2.13	3.37	3.00
	100	1.86	2.03	3.58	3.54	5.25	5.31
	1000	10.02	9.81	18.43	16.80	26.31	24.26
	1800	7.49	7.31	13.84	12.54	18.96	17.94
Italy	10	3.00	3.00	4.50	4.50	4.50	4.5
	100	4.00	4.03	7.82	7.68	11.00	10.79
	1000	7.71	7.97	15.54	15.65	21.18	22.30
	1800	12.22	11.72	22.98	20.57	28.60	30.38

In Table 9 we report '–' in those cases where a parallel solution was not obtained within the timeout. In the 10 seconds time limit scenario, we observe a speedup factor close to 1 for the UK and Italy. The speed up factor improves considerably after 1000 seconds. It goes up to a factor of 26 for the UK (IS with 12 cores and 1800 secs) to a factor of 30 with 8 cores (RC with 12 cores and 1800 secs).

9 Conclusions and Future Work

We have presented an efficient local search algorithm for solving the Edge-Disjoint Rooted Distance-Constrained Minimum Spanning-Trees problem. We presented two move operators along with their complexities and an incremental evaluation of the neighbourhood and the objective function. Furthermore, we have proposed a parallelisation scheme for the local search algorithm, which significantly reduces the time required by sequential version to reach high quality solutions. Any problem involving tree structures could benefit from these ideas and the techniques presented are relevant for a constraint-based local search framework where this type of incrementality is needed for network design problems. The effectiveness of our approach is demonstrated by experimenting with a set of problem instances taken from real-world long-reach passive optical network deployments in Ireland, Italy, and the UK.

In the future we would like to extend ERDCMST with the notion of optional nodes, since this extension is a common requirement in several applications of ERDCMST. Effectively this means that we would compute for every facility a Minimum Steiner Tree where all clients are covered but the path to them may follow some optional nodes. We also plan to investigate alternative heuristics for selecting the most suitable node and subtree for deletion at each iteration of the local search algorithm.

Acknowledgments

This work was supported by DISCUS (FP7 Grant Agreement 318137), and Science Foundation Ireland (SF) Grant No. 10/CE/I1853. The Insight Centre for Data Analytics is also supported by SFI under Grant Number SFI/12/RC/2289.

References

1. Arbelaez, A., Codognet, P.: Massivelly Parallel Local Search for SAT. In: ICTAI'12, pp. 57–64. IEEE Computer Society, Athens, Greece (2012)
2. Arbelaez, A., Codognet, P.: From sequential to parallel local search for SAT. In: 13th European Conference on Evolutionary Computation in Combinatorial Optimisation (EvoCOP'13), pp. 157–168 (2013)
3. Arbelaez, A., Codognet, P.: A survey of parallel local search for sat. In: Theory, Implementation, and Applications of SAT Technology. Workshop at JSAI'13. Toyama, Japan (2013)
4. Arbelaez, A., Hamadi, Y.: Improving parallel local search for SAT. In: LION 5, pp. 46–60 (2011)
5. Arbelaez, A., Mehta, D., O'Sullivan, B., Quesada, L.: Constraint-based local search for the distance-and capacity-bounded network design problem. In: ICTAI'14, pp. 178–185. IEEE (2014)
6. Arbelaez, A., Mehta, D., O'Sullivan, B., Quesada, L.: A constraint-based parallel local search for disjoint rooted distance-constrained minimum spanning tree problem. In: Workshop on Parallel Methods for Search & Optimization (2014)
7. Arbelaez, A., Mehta, D., O'Sullivan, B., Quesada, L.: A constraint-based local search for edge disjoint rooted distance-constrained minimum spanning tree problem. In: L. Michel (ed.) CPAIOR'15, *Lecture Notes in Computer Science*, vol. 9075, pp. 31–46. Springer (2015). DOI 10.1007/978-3-319-18008-3_3. URL `http://dx.doi.org/10.1007/978-3-319-18008-3_3`
8. Baraglia, R., Hidalgo, J.I., Perego, R.: A parallel hybrid heuristic for the TSP. In: EvoWorkshops, pp. 193–202 (2001)
9. Caniou, Y., Diaz, D., Richoux, F., Codognet, P., Abreu, S.: Performance analysis of parallel constraint-based local search. In: PPOPP, pp. 337–338 (2012)
10. Crainic, T.G., Gendreau, M.: Cooperative parallel tabu search for capacitated network design. J. Heuristics **8**(6), 601–627 (2002)
11. Davey, R., Grossman, D., Rasztovits-Wiech, M., Payne, D., Nesset, D., Kelly, A., Rafel, A., Appathurai, S., Yang, S.H.: Long-reach passive optical networks. Journal of Lightwave Technology **27**(3), 273–291 (2009)
12. Dung, P.Q., Deville, Y., Van Hentenryck, P.: Constraint-based local search for constrained optimum paths problems. In: CPAIOR, pp. 267–281 (2010)
13. Ho, J.M., Lee, D.T.: Bounded diameter minimum spanning trees and related problems. In: Proceedings of the fifth annual symposium on Computational geometry, SCG '89, pp. 276–282. ACM, New York, NY, USA (1989). DOI 10.1145/73833.73864. URL `http://doi.acm.org/10.1145/73833.73864`
14. Hoos, H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann (2005)
15. Hunter, D.K., Lu, Z., Gilfedder, T.H.: Protection of long-reach PON traffic through router database synchronization. Journal of Optical Communications and Networking **6**(5), 535–549 (2007)
16. Leitner, M., Ruthmair, M., Raidl, G.R.: Stabilized branch-and-price for the rooted delay-constrained steiner tree problem. In: J. Pahl, T. Reiners, S. Vo (eds.) INOC, *Lecture Notes in Computer Science*, vol. 6701, pp. 124–138. Springer (2011)
17. Martins, R., Manquinho, V.M., Lynce, I.: An overview of parallel SAT solving. Constraints **17**(3), 304–347 (2012)
18. Michel, L., See, A., Van Hentenryck, P.: Parallel and distributed local search in comet. Computers and Operations Research **36**, 2357–2375 (2009)

19. Oh, J., Pyo, I., Pedram, M.: Constructing minimal spanning/steiner trees with bounded path length. Integration **22**(1-2), 137–163 (1997)
20. Payne, D.B.: FTTP deployment options and economic challenges. In: Proceedings of the 36th European Conference and Exhibition on Optical Communication (ECOC 2009) (2009)
21. Roli, A.: Criticality and Parallelism in Structured SAT Instances. In: P.V. Hentenryck (ed.) CP'02, *LNCS*, vol. 2470, pp. 714–719. Springer, Ithaca, NY, USA (2002)
22. Ruthmair, M., Raidl, G.R.: A kruskal-based heuristic for the rooted delay-constrained minimum spanning tree problem. In: R. Moreno-Díaz, F. Pichler, A. Quesada-Arencibia (eds.) EUROCAST, *Lecture Notes in Computer Science*, vol. 5717, pp. 713–720. Springer (2009)
23. Ruthmair, M., Raidl, G.R.: Variable neighborhood search and ant colony optimization for the rooted delay-constrained minimum spanning tree problem. In: R. Schaefer, C. Cotta, J. Kolodziej, G. Rudolph (eds.) PPSN (2), *Lecture Notes in Computer Science*, vol. 6239, pp. 391–400. Springer (2010)
24. Salama, H.F., Reeves, D.S., Viniotis, Y.: The delay-constrained minimum spanning tree problem. In: ISCC, pp. 699–703 (1997)
25. Shylo, O.V., Middelkoop, T., Pardalos, P.M.: Restart Strategies in Optimization: Parallel and Serial Cases. Parallel Computing **37**(1), 60–68 (2011)
26. Truchet, C., Arbelaez, A., Richoux, F., Codognet, P.: Estimating parallel runtimes for randomized algorithms in constraint solving. To Appear in Journal of Heuristics (2015)
27. Van Hentenryck, P., Michel, L.: Constraint-based local search. The MIT Press (2009)
28. Verhoeven, M., Aarts, E.: Parallel local search. Journal of Heuristics **1**(1), 43–65 (1995)
29. Verhoeven, M., Severens, M.: Parallel local search for steiner trees in graphs. Annals of Operations Research **90**, 185–202 (1999)

Appendix

The node and subtree operators are general enough to tackle both RDCMST and ERDCMST. In this appendix we provide experimental results of the local search approach and dedicated algorithms for the RDCMST problem.

PBH and KBH [22] are two dedicated heuristics to tackle the RDCMST. The former, inspired in the Prim's algorithm, iteratively adds nodes in the solution using the cheapest edge connecting the node and the current solution. The latter, inspired in the Kruskal algorithm, starts by sorting the list of edges and iteratively adds an edge if no cycles is created in the solution. Additionally, the authors perform local moves to improve the solution, these moves rely in a pre-computed backup route from the root node to any node. We recall that these local moves cannot be applied to the ERDCMST as the pre-computed route from the root node to a certain node might not be available due to the disjoint constraint.

BKRUS [19] is a well-known heuristic for solving RDCMST the cost function and the delay function in RDCMST are equivalent and follow the Triangle inequality property (i.e., given three points a, b, and c then $d(a,b) + d(b,c) > d(a,c)$). The overall complexity of BKRUS is cubic in terms of number of nodes, which hinders its scalability when the bound on the path-length is tight. Although there are approaches that are better than BKRUS [16, 23], our aim is not to claim superiority over the RDCMST approaches since we are solving a more general problem for which these approaches would not be applicable. Instead, our aim is to show that although our approach is not specialised for RDCMST, it still provides very good quality solutions.

9.1 Experiments for the RDCMST problem

We experimented with the following two set of instances for the RDCMST problem:

- *Real-life*: In this scenario we report results on a set of real-life instances combing from our industrial partner in Ireland, each instance contains $|E| \in \{200, 300, \ldots, 800\}$ clients, and for each instance we report the median time across 11 executions with 10 minutes cutoff for the subtree operator.
- *Random* In this scenario we consider two sets of instances from [22].[5] Each set ($|U| \in \{500, 1000\}$) contains 30 complete graphs with integer edge cost and lengths uniformly distributed in [1,99]. Similar to the work of [22] each instance was executed 30 times with 10 minutes cutoff for the subtree operator.

[5] All instances are available at `https://www.ads.tuwien.ac.at/w/Research/Problem_Instances`

We limit our attention for the PBH and KBH to random instances due to these two algorithms are not available online.[6] The random set of instances does not satisfy the Triangle inequality property required for BKRUS and therefore for this dataset we only consider CPLEX, PBH, KBH, and our CBLS algorithm. For the real-life instances we consider CPLEX, BKRUS, and our CBLS algorithm.

To generate the initial solution for the CBLS algorithm in these experiments we used a similar approach as [22] by iteratively adding nodes in the solution using Prim's algorithm, and for nodes violating the distance constraint we use the shortest path from the root to the node to reconnect the node and generate a valid initial solution.

Table 10 reports detailed results of the cost and GAP percentage with respect to the lower bound. Here we observe that the GAP for CPLEX increases considerably as the number of clients increases. On the other hand, for the difficult case (λ=415) LS (subtree operator) reports better upper bounds than CPLEX and BKRUS in all 7 instances, and the quality of the solution with respect to the lower bound does not degrade with the problem size.

Table 11 shows results of the second experimental scenario with random instances. In this table we report the results for different length limits ($\lambda \in \{6, 10, 20, 40\}$) for two dedicated algorithms for the RDCMST problem: PBH (Prim's based algorithm [24]) and KBH (Kruskal's based algorithm [22]). Due to the subtree operator usually outperforms the node operator, in this experiments we focus our attention to the former one. We would like to recall that our claim is not to be superior over existing dedicated algorithms for the RDCMST, instead we propose a generic constraint-based local search framework where adding more constraints is a straightforward process.

In this experiment we observe that LS reports a close performance to the dedicated algorithms, and the solution quality (with respect to the best between PBH and KBH) is between 2.6% - 4.7% for $|E| = 500$ and between 12.7% - 21.9% for $|E| = 10000$. Indeed, the local search algorithm outperforms at least one of the dedicated algorithm in four out of the eight random scenarios. As pointed out in [22] the runtime of the dedicated algorithms is on average up to 3 minutes, while we use 10 minutes for each experiment. An important part of the time reduction in [22] consists in removing an important number of edges (up to 88% of the original problem for these instances) in a pre-processing phase. In this paper we omit pre-processing since an important number of the edges can not be removed as they might be relevant to satisfy other constraints (e.g., disjointness).

[6] We would like to thank Mario Ruthmair, author of PBH and KBH, for sharing the instances used in [22] and for pointing out that the source of the algorithms can not be distributed due to license issues.

Table 10 Results for RDCMST instances from the Irish dataset.

λ	$\|U\|$	LB		Node	Subtree	CPLEX	BKRUS
415	200	2183	Cost	2653	2445	2599	2699
			Gap	17.72	10.71	16.00	19.11
	300	3019	Cost	3584	3364	4365	3789
			Gap	15.77	10.25	30.83	20.34
	400	3576	Cost	4316	3953	5618	4498
			Gap	17.15	9.54	36.35	20.50
	500	4123	Cost	5234	4507	6642	5457
			Gap	21.21	8.51	37.92	24.44
	600	4643	Cost	6158	5020	7542	6143
			Gap	24.59	7.50	38.43	24.41
	700	5130	Cost	6896	5545	8737	6303
			Gap	25.60	7.47	41.28	18.60
	800	5619	Cost	7896	6123	10744	6835
			Gap	28.84	8.23	47.70	17.79
623	200	2183	Cost	2402	2293	2333	2328
			Gap	9.12	4.81	6.42	6.22
	300	2980	Cost	3183	3118	3446	3194
			Gap	6.38	4.41	13.51	6.68
	400	3560	Cost	3731	3729	4448	3960
			Gap	4.58	4.54	19.97	10.09
	500	4099	Cost	4555	4282	5196	4524
			Gap	10.02	4.27	21.11	9.40
	600	4638	Cost	5114	4814	5603	5081
			Gap	9.31	3.66	17.21	8.71
	700	5122	Cost	5745	5306	5957	5459
			Gap	10.85	3.48	14.01	6.18
	800	5609	Cost	6616	5796	17830	6061
			Gap	15.22	3.22	68.54	7.45

Table 11 Results for RDCMST random instances from [22] for two dedicated algorithms and the proposed LS algorithm using the subtree operators. Highlighted cells indicate that subtree is better that at least one of the two dedicated algorithms.

λ	$\|U\|$=500			$\|U\|$=1000		
	PBH	KBH	Subtree	PBH	KBH	Subtree
6	9340	9067	**9257**	10858	9942	12307
10	4975	4421	**4643**	5715	5040	6461
20	2247	2124	**2203**	2579	3291	**3025**
40	1167	1155	1186	1491	1486	1704