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

Alejandro Arbelaez, Deepak Mehta, and Barry O'Sullivan and Luis Quesada

INSIGHT Centre for Data Analytics
University College Cork, Ireland
alejandro.arbelaez@insight-centre.org

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 modelled as Rooted Distance-Constrained Minimum Spanning-Tree Problem (RDCMST), which is NP-hard. These networks 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 Disjoint RDCMST (DRDCMST). We present a constraint-based parallel local search algorithm for solving DRDCMST. A traditional way of extending a sequential algorithm to run in parallel is to either perform portfolio-based search in parallel or to perform 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 any other problem. 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 and the UK. Results show that performing moves in parallel can significantly reduce the time required by our local-search approach.

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 is connected to the set of exchange-sites via optical fibres, the length of the fibre between an exchange-site and its metro-node is bounded due to signal loss, and the goal is to minimise the cost resulting from the total length of fibres [1]. In VLSI circuit design path delay is a function of maximum interconnection path length while power consumption is a function of the total interconnection length [2]. In package shipment service 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 [3].

Many of these network design problems can be modelled as Rooted Distance-Constrained Minimum Spanning-Tree Problem (RDCMST) [2] 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 Disjoint Rooted Distance-Constrained Minimum Spanning-Trees Problem (DRDCMST). 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 [4, 5] have focused on dedicated algorithms which are hard to extend with side constraints, and therefore these algorithms cannot be extended for solving DRDCMST. We present 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 objective function which is often a key element for efficient local search algorithms. Our local search algorithm is able to solve both RDCMST and DRDCMST problems. We then extend our sequential algorithm to perform search in parallel. Traditional way of extending a sequential algorithm to run in parallel is to either perform portfolio-based search in parallel or to perform 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 effectiveness of our approach is demonstrated by experimenting with a set of problem instances taken from real-world passive optical network deployments in Ireland and the UK. 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 M be the set of facilities. Let E be the set of clients. Let $E_i \subseteq E$ be the set of clients that are associated with facility $m_i \in M$. We use N to denote the set of nodes, which is equal to $M \cup E$. We use T_i to denote the tree network associated with facility i. We also use $N_i \subseteq N = E_i \cup \{m_i\}$ to denote the set of nodes in T_i. Let λ be the maximum path-length from a facility to any of its clients.

Rooted Distance-Constrained Minimum Spanning-Tree Problem (RDCMST). Given a facility $m_i \in M$, the set of clients E_i, a set of feasible links $L_i \subseteq N_i^2$, two real number, a cost c_l and a distance d_l for each link $l \in L_i$, and a real number λ, the RDCMST to find a spanning tree T_i with minimum total cost such that the length of the path from the facility m_i to any $e_j \in E_i$ is not greater than λ.

Disjoint Rooted Distance-Constrained Minimum Spanning-Trees Problem (DRD-CMST). Given a set of facilities M, a set of clients E, a set of feasible links $L \subseteq N^2$, two real number, a cost c_l and a distance d_l for each link $l \in L$, an association of clients with two facilities $\pi : E \rightarrow M^2$, and a real number λ, the DRDCMST is to find a spanning tree T_i for each facility m_i such that:

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

Complexity. DRDCMST involves finding a rooted distance-bounded spanning tree for every facility whose total cost minimum. This problem is known to be NP-complete [2].

3 Iterated Constraint-based Local Search

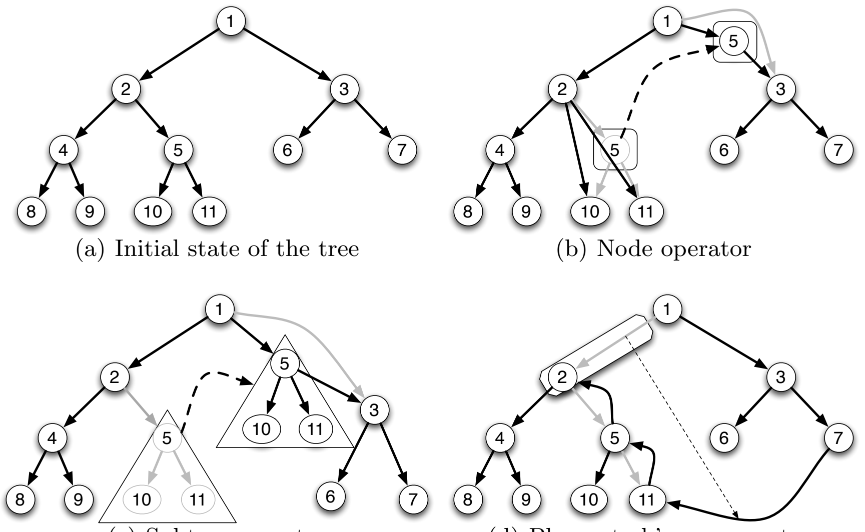
The Iterated Constraint-based Local Search (ICBLS) [8, 9] framework depicted in Algorithm 1 comprises two phases. First, in a local search phase, the algorithm improves the current solution, little by little, by performing small changes. Generally speaking, it employs a move operator in order to move 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 scape from difficult regions of the search (e.g., a local minima). Finally, the acceptance criterion decides whether to update s^* or not. To this end, with a probability 5% s'^* will be chosen, and the better one otherwise.

Our algorithm starts with a given initial solution where all clients are able to reach their facilities while satisfying all constraints (i.e., the upper bound in the length and disjointness). We switch from the local search phase to perturbation when a local minima is observed; in the perturbation phase we perform a given number of random moves (20 in this paper).

The stopping criteria is either a timeout or a given number of iterations.

Algorithm 1 Iterated Constraint-Based Local Search

```
1: $s_0$ := Initial Solution
2: $s^*$ := ConstraintBasedLocalSearch($s_0$)
3: **repeat**
4:     $s'$ := Perturbation($s^*$)
5:     $s'^*$ := ConstraintBasedLocalSearch($s'$)
6:     $s^*$ := AcceptanceCriterion($s^*$, $s'^*$)
7: **until** No stopping criterion is met
```



(a) Initial state of the tree (b) Node operator

(c) Subtree operator (d) Pham et al.'s arc operator

Fig. 1. Move operators

3.1 Move-Operators

In this section we propose two new move operators. We use T_i to denote the tree associated with facility i. An edge between two clients e_p and e_q is denoted by $\langle e_p, e_q \rangle$.

Node operator (Figure 1(b)) moves a given node e_i from the current location to another in the tree. As a result of this, all successors of e_i will be directly connected to the predecessor node of e_i. e_i can be placed as a new successor for another node or in the middle of an existing arc in the tree.

Subtree operator (Figure 1(c)) moves a given node e_i and the subtree emanating from e_i from the current location to another in the tree. As a result of this, the predecessor of e_i is not connected to e_i, and all successors of e_i are still directly connected to e_i. e_i can be placed as a new successor for another node or in the middle of an existing arc.

Arc Operator (Figure1(d)). In this paper we limit our attention to moving a node or a complete subtree. [10] proposed to move arcs in the context of the Constrained Optimum Path problems. Pham et al. move operator (Figure 1(d)) chooses an arc in the tree and finds another location for it without breaking the flow.

3.2 Operations and Complexities

We first present node and subtree operators as they share similar features. For an efficient implementation of the move operators, it is necessary to maintain f^i_j (the length of the path from facility i to client j) and b^i_j (length of the path from e_j down to the farthest leaf associated to it in tree T_i) for each client e_i associated with each facility m_i. This information will be used to maintain the path-length constraint. Let e_{p_j} be the immediate predecessor of e_j and let S_j be the set of immediate successors of e_j in T_i. Table 1 summarises the complexities of the move operators, in this table n denotes the total number of clients associated to one facility.

Table 1. Complexities of different operations

	Node	Subtree	Arc
Delete	$O(n)$	$O(n)$	$O(1)$
Feasibility	$O(1)$	$O(1)$	$O(n)$
Move	$O(n)$	$O(n)$	$O(n)$
Best move	$O(n)$	$O(n)$	$O(n^3)$

Delete. Deleting a node e_j from T_i requires a linear complexity w.r.t. the number of clients of 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 client e_{p_j} in T_i. In addition, the node operator updates $f^i_{j'}$ for all the nodes j' in the subtree emanating from e_i. After deleting a node e_j or a subtree emanating from e_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 e_j and reconnecting them to e_{p_j}.

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

Feasibility. Checking feasibility for a move can be performed in linear time by using f^i_j and b^i_j. If e_j is inserted between an arc $\langle e_p, e_q \rangle$ then we check the following:

$$f^i_p + c_{pj} + c_{jq} + b^i_q < \lambda$$

If e_p is a leaf-node in the tree and e_j is placed as its successor then the following is checked:

$$f^i_p + c_{pj} + b^i_j < \lambda \qquad (1)$$

Move. A move can be performed in linear time. We recall that this move operator might replace an existing arc $\langle e_p, e_q\rangle$ with two new arcs $\langle e_p, e_j\rangle$ and $\langle e_j, e_q\rangle$. This operation requires to update f^i_j for all nodes in the emanating tree of e_j, and b^i_j in all nodes in the path from the facility acting as a root node down to the new location of e_j. The objective function must be updated as follows:

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

Best Move. Selecting the best move involves traversing all clients associated with the facility and selecting the one with the maximum reduction in the objective function.

Now we switch our attention to the arc operator. This operator does not benefit by using b^i_j. The reason is that moving a given arc from one location to another requires changing the direction of a certain number of arcs in the tree. Deleting an arc requires constant complexity, this operation generates two separated subtrees and no data structures need to be updated. Checking the feasibility of adding an arc $\langle e_{p'}, e_{q'}\rangle$ to connect two subtress requires linear complexity. It is necessary to actually traverse the new tree to obtain the distance from $e_{q'}$ to the farthest leaf in tree of facility i. Performing a move requires a linear complexity, and it involves updating f^i_j for the new emanating tree of $e_{q'}$. And performing the best move requires a cubic time complexity, the number of possible moves is n^2 (total number of possible arcs for connecting the two subtree) and for each possible move it is necessary to check feasibility. Due to the high complexity ($O(n^3)$) of the arc operator to complete a move, hereafter we limit our attention to the node and subtree operators.

Disjointness To ensure disjointness among spanning-trees we maintain a $2 \cdot |E|$ Matrix, where $|E|$ represents the number of clients. For every client, there are two integers indicating the predecessor in the primary and secondary facilities, these two numbers must always be different.

3.3 Sequential Algorithm

The pseudo-code for constraint-based local search is depicted in Algorithm 2. It starts by selecting a client e_j of a facility m_i randomly and performs a move by using one of the move operators as described before. Here the move operator, which is itself a function, is passed as a parameter. In each iteration of the algorithm (Lines 9-19), we identify the best location for e_j. A location in a tree T_i is defined by (e_q, S) where e_q denotes the parent of e_j and S denotes the set of successors of e_j after the move is performed. Broadly speaking, there are two options for the new location: (1) Breaking an arc $\langle e_p, e_q\rangle$ and inserting e_j in between them such that the parent node of e_j would be e_p and the successor set

Algorithm 2 ConstraintBasedLocalSearch (*move-op*,*sol*)

```
1: ${T_1 \dots T_n} \leftarrow sol$
2: $list \leftarrow \{(m_i, e_j)|m_i \in M \wedge e_j \in E_i\}$
3: $fcg \leftarrow \{(m_i, m_j)||N_i \cap N_j| \geq 2\}$
4: while $list \neq \emptyset$ do
5:     Select $(m_i, e_j)$ randomly from $list$
6:     $Best \leftarrow \{(e_{p_j}, S_j)\}$
7:     Delete $e_j$ from $T_i$ and update $T_i$
8:     $cost \leftarrow \infty$
9:     for $(e_q, S)$ in $Locations(move\text{-}op, T_i)$ do
10:        if FeasibleMove($move\text{-}op$, $(e_q, S), e_j$) then
11:            $cost' \leftarrow$ CostMove( $(e_q, S)$, $e_i$)
12:            if $cost' = cost$ then
13:                $Best \leftarrow Best \cup \{(e_q, S)\}$
14:            else if $cost' < cost$ then
15:                $Best \leftarrow \{(e_q, S)\}$
16:                $cost \leftarrow cost'$
17:            end if
18:        end if
19:    end for
20:    Select $(e_{q'}, S')$ randomly from $Best$
21:    if $e_{q'} \neq e_{p_j} \vee S' \neq S_j$ then
22:        $list \leftarrow list \cup \{(m_k, e)|(m_i, m_k) \in fcg \wedge e \in N_k\}$
23:    end if
24:    $list \leftarrow list - \{(m_i, e_j)\}$
25:    $T_i \leftarrow$Move( $T_i$,$move\text{-}op$, $(e_{q'}, S')$, $e_j$ )
26: end while
27: return $\{T_1 \dots T_n\}$
```

would be singleton, i.e., $S = \{e_q\}$; (2) Adding a new arc $\langle e_p, e_j \rangle$ in the tree in which case the parent of e_j is e_p and $S = \emptyset$. *Locations* returns all the locations relevant w.r.t. a given move-operator. Line 10 verifies that the new move is not breaking any constraint and *CostMove* returns the cost of applying such move using a given move operator.

Instead of verifying that the local minima is reached by exhaustively checking all moves for all clients of all facilities, we use a *facility connectivity graph* (*fcg*) where the vertices represent trees (associated with facilities) and an edge between them represent that a change in one tree can affect the change in another tree. Notice that a change in a tree of a facility is restricted by another tree of another facility if they share at least 2 clients because an edge between them can appear in at most one tree. Consequently, an edge between two facilities is added (Line 25) if they share at least two clients. Therefore, all affected facilities in the *fcg* will be added in to *list* for testing them again in upcoming iterations. Otherwise, when no improvement is observed, the current node is removed from *list*. A local minima, i.e., no improvements in the objective can be obtained with the current solution is reached once *list* is empty. This mechanism helps in reducing the time significantly by reducing the number of useless moves. Algorithm 2 can tackle both RDCMST and DRDCMST. For the former, there would be only one facility. In addition, *FeasibleMove* will only check path-length constraint.

4 Parallel Algorithm

Parallelization has been widely studied to speed-up and improve performance of local search algorithms to tackle a large variety of problems including: TSP [11], Capacited Network Design [12], Steiner Tree [13], SAT [14], and CSPs [15] just to name a few. These approaches employ the Multi-walk and/or Single-walk framework [16] to devise the parallel algorithm. In particular we focus our attention in the constraint-based local search solvers.

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 search space.

The multi-walk method has two important properties. First, no load balancing is required to parallelise the sequential algorithm. Second, in theory, it is possible to reach linear and super-linear speedups [17], however, in practice the speedup of traditional local search algorithm is far from linear and, it is usually limited to a few number of cores [18].

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

In the context of SAT we observe two different levels of parallelism (see [19] for a recent survey). On the one hand, multi-walk approaches execute multiple algorithms at the same time with or without cooperation [18]. In the cooperative framework processes exchange the best solution with other processes in order to properly craft a new starting point. On the other hand, single-walk approaches aim at flipping multiple variables at the same time [20]

In [15] the authors exploit the multi-walk framework to parallelize the *Adaptive Search* library, a constraint-based local search library. Experiments on a set of academic benchmarks indicate that the speed-up varies from problem to problem, and under some particular circumstances linear-speedups can be obtained up to 8000 processes. [21] describes the parallel architecture of the comet solver, a robust local search solver, the architecture involves abstractions for implementing mutl-walk (with and without cooperation) and single-walk solvers. And [22] studies the use of GPUs to speedup the resolution process of a generic constraint-based local search algorithm in a GPU.

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

```
1: $S := \{ \}$
2: while $fcg$ is not empty and $|S| < maxElements$ do
3:     $v :=$ random vertex in $fcg$
4:     $S := S \cup v$
5:     Remove $v$ and its neighbours from $fcg$
6: end while
7: return $S$
```

4.1 Constraint-based Parallel Local Search for DRDCMST

In this paper, 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 DRDCMST can 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 constraints are violated. We use this approach to develop a parallel algorithm for the DRDCMST problem.

Let L_{ij} be the set of all locations for a node associated with client e_j where it can be moved in the tree T_i associated with the facility m_i by using either node or sub-tree move-operator. 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.

Now finding a set of nodes from a same tree whose sets of locations are pair-wise mutually exclusive is more difficult as there would be greater degree of overlapping depending on the input graph. As the sets of locations for the selected nodes must be independent in order to avoid the cost of communication and sharing resources between parallel processes we select at most one node from one tree. Therefore, the number of moves that can be performed simultaneously is bounded by the number of facilities. Recall that changing the location of a node within a tree is not only constrained by the other nodes of the same tree but also by the nodes of other trees because of the disjoint constraint. In order to determine the number of facilities, 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 each client would have at most one predecessor, all elements in the set can be 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. The degree of parallelism obtained using this approach is dominated by the cardinality of the maximum independent set in *fcg*, in practice we expect sparse graphs in real networks, therefore the cardinality of independent sets should be more than a few tens of elements.

2. **Random conflict** selects, uniformly at random, n facilities and resolve the conflicts between clients apriori. It is recalled that two facilities can be in conflict if and only if they share at lest two clients. Let us say that two facilities m_i and m_i' are selected, and the clients e_j and e_j' are connected to both facilities. Let $C = L_{ij} \cap L_{i'j'}$ be a non-empty set. This implies that L_{ij} and $L_{i'j'}$ are not mutually exclusive. To resolve the conflict we modify the sets L_{ij} and $L_{i'j'}$ such that they become mutually exclusive.
   - If $e_k \in C$ is already connected to e_j' in T_i then we remove e_k from $L_{i'j'}$, or vice-versa.
   - If $e_k \in C$ is not connected to any of e_j or e_j' in both trees T_i and T_i' then we remove e_k randomly from either L_{ij} or $L_{i'j'}$.

   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.

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

```
1: $s$ := Initial Solution
2: repeat
3:     $P$ := CreatePartition($s$)
4:     for each $s_i \in P$ do in parallel
5:         while local time limit $t$ for parallelism has not been reached do
6:             if $s_i$ is internally in a local minima then
7:                 $s_i^*$ := Perturbation($s_i$)
8:             end if
9:             $s_i^{'*}$ := ConstraintBasedLocalSearch(move-op,$s_i^*$ )
10:            $s_i$ := AcceptanceCriterion($s_i^*$, $s_i^{'*}$)
11:        end while
12:    end parfor
13: until
```

The Iterated Constraint-based Parallel Local Search algorithm (ICPLS) works in two phases. First, the algorithm selects a set of facilities, denoted by P. If the set is independent then the locations of the clients of different facilities are mutually exclusive. If the set is in conflict then the locations of the clients of different facilities are modified by restricting their locations to resolve any conflicts. Second, for each facility $p_i \in P$ it performs, in parallel, the sequential local search algorithm for a given amount of time t to explore the search space. As the parallel algorithm invokes the constraint-based local search algorithm depicted in the previous section, we would like to differentiate two plateaus. A local minima of the problem, i.e., a state in which no neighbours solutions yield to improvements in the objective (used in the sequential algorithm) and a local minima in a tree, i.e., when a single tree is internally in a local minima but the global picture of the whole problem is still unknown.

Informally speaking, the sequential algorithm scans the list of active clients (*list*) and clients are deleted from the list whenever they cannot improve the objective function. And perturbation starts when a local minima in the problem is reached (i.e., *list* $= \{ \}$). Maintaining the same scheme in the parallel algorithm might introduce an important processors idle-time, in particular when approaching to a local minima. Therefore, in order to minimise idle-time, we start perturbation locally for each tree as soon as an internal local minima for a given tree is reached. In addition, in order to reduce synchronisation among processors, after applying a move in a tree only nodes of the same tree will be added into *list* (line 22 in Algorithm 2). In this schema the global solution of the problem is the aggregation of independent solutions obtained for individual trees.

Algorithm 4 depicts the ICPLS proposed in this paper. Similarly to the sequential algorithm, the algorithm starts with an initial solution, then it computes P using *independent set* or *random conflict*, notice that CreatePartitions receives the current solution s in order to decide whether it is necessary to break conflicts (e.g., when using independent set). In the parallel section of the algorithm, we verify if the tree associated to p_i is internally in a local minima to perturb the solution (lines 6-8). Afterwards, the Constraint-based Local Search procedure is invoked, and the same acceptance criterion as the sequential algorithm is also invoked. It is worth noticing that unlike the sequential algorithm, where the a local minima is strictly reached after invoking local search, in the parallel algorithm it might be the case that due to the local time limit for each parallel execution, the parallel algorithm has not reached the local minima.

5 Long-Reach Passive Optical Networks

We now describe a real-world application whose instances are used for evaluating our approach. Long-Reach Passive Optical Networks (LR-PONs) provide an economically viable solution for fibre-to-the-home network architectures [1]. In LR-PON fibres are distributed from the Metro-Nodes (MNs) to the Exchange-Sites (ESs) through cables that forms a tree distribution network. A major fault occurrence in LR-PON would be a complete failure of the MN, which could affect tens of thousands of customers. The dual homing protection mechanism for LR-PON enables customers to be connected to two MNs, so that whenever a single MN fails all customers are still connected to a back-up. Notice that the paths from an ES to its two MNs cannot contain the same link. Otherwise, this would void the purpose of having two MNs. Given as association of MNs with ESs the problem is to determine the routes of cables such that there are two edge-disjoint paths from an ES to its two MNs, the length of each path is below threshold and the total cable length required for connecting each ES to two MNs

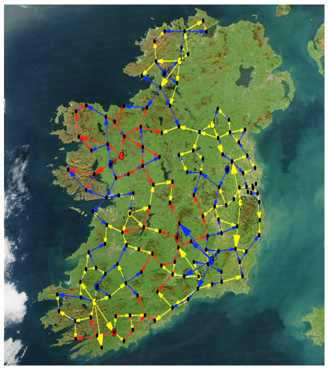


Fig. 2. Example of a LR-PON network for Ireland where each exchange-site is connected to two metro-nodes through disjoint paths. In the plot the subnetwork of each metro node is associated with a colour. Two subnetworks may have the same colour if they do not share nodes.

is minimised. Notice that here metro-nodes are facilities and exchange-sites are clients.

6 Empirical Evaluation

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.

To evaluate the performance of the proposed parallel local search algorithm, we use two datasets corresponding to real networks from Ireland, with 1122 exchange sites and 18, 20, 22, 24 Metro Nodes; and the UK, with 5394 exchange sites and 75, 80, 85, 90 Metro Nodes. In preliminary experiments we observed that the subtree operator outperformed the node operator, for this reason hereafter we limit our attention to the subtree operator.

The goal of the parallel algorithm is twofold. First, decreasing the elapsed time for finding a target solution (i.e., optimal or near-optimal solution). Second, the quality of the solution might be better when increasing the processing power (i.e., adding more cores and processors).

Table 2 shows the results of the empirical evaluation of the parallel algorithm. In this table we present the cost of the solution for the sequential algorithm, the parallel algorithms, and the lower bound (LB) of the solution obtained using CPLEX. We use the multi-walk (MW), i.e., executing multiple copies of the algorithm with different random seeds, framework as a baseline for comparison; we also include the proposed parallel algorithms using both *random conflict* (RC) and *independent set* (IS) for partitioning the problem. For each experiment we report the median value across 11 executions with a time-limit of 10 minutes for each experiment. Figure 3 (top) shows the evolution of the solution to solve an instance form the Irish dataset (with 18 metro nodes) and another from the UK dataset (with 75 metro nodes) for a typical execution[1]. The x-axis gives the quality of the solution of the sequential and parallel algorithms.

Country	$\|M\|$	Seq	LB	Number of Cores								
				4			8			12		
				MW	RC	IS	MW	RC	IS	MW	RC	IS
	18	17255	14809	17263	**17194**	17208	17254	**17193**	17201	17254	17189	**17182**
Ireland	20	16987	14845	16976	**16947**	16958	16982	**16926**	16951	16981	16940	**16922**
$\|E\|$=1121	22	16835	14990	**16799**	16813	16812	16810	16819	**16798**	16810	16819	**16785**
	24	16233	14570	**16224**	16257	16247	**16235**	16255	16251	**16238**	16280	16275
	75	67085	54720	67135	65078	**65075**	67111	64992	**64903**	67135	**64873**	64890
UK	80	66178	54975	66254	64371	**64305**	66247	64211	**64180**	66247	**64111**	64147
$\|E\|$=5394	85	65096	55087	65094	63533	**63517**	65011	63425	**63409**	65011	**63351**	63381
	90	63528	55035	63528	62310	**62251**	63535	62170	**62149**	63551	62150	**62122**

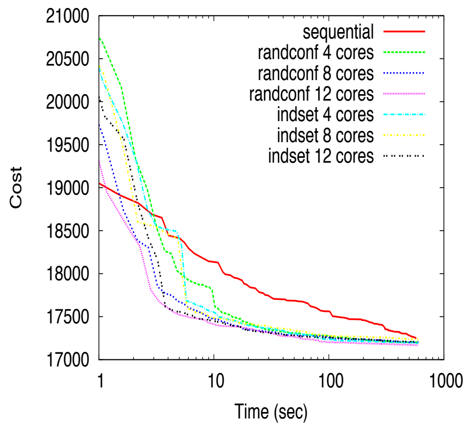
Table 2. Performance summary with a 10-minute time-out for each experiment for the sequential algorithm (Seq), the lower bound (LB) obtained using CPLEX, and the parallel versions using the Multi-walk (MW), Single-walk + *random conflict* (RC) and Single-walk + *independent set* (MW+IS) architectures.

The sequential algorithm finds a very good quality solution within the time limit for the instances of Ireland with a GAP of up to 10% with respect to the lower bound. For this reason, when increasing the number of cores we observe very little difference in the performance of the algorithms. Nevertheless, it can be observed that the proposed parallel algorithms outperform the base line in 6, 7 and 7 instances using 4, 8, and 12 cores respectively out of 8 instances. When analysing Figures 3(a) and 3(b) we observe an important GAP between the parallel solvers and the sequential one in the progress of the solver. During the first few seconds the sequential algorithm dominates the performance, that is because, the parallel algorithm selects a subset of n metro nodes (where n is the number of parallel processes) and only improves the solution for these metro nodes for a given amount of time (see Algorithm 4). However, the parallel solvers dominate performance after a few seconds. Moreover, using more cores reduces the elapsed time to reach good quality solutions.

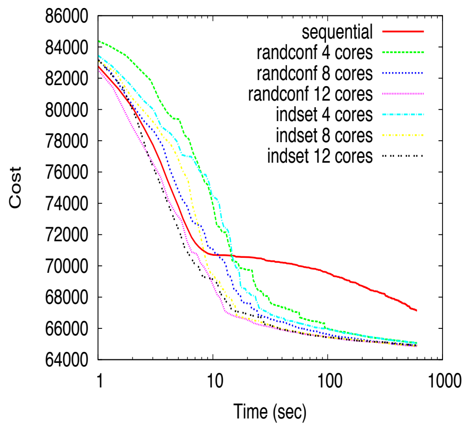
On the other hand, for UK instances, which are about four times bigger (w.r.t. number of exchange sites), except for the multi-walk approach (where no significance improvement is seen), we observe that the parallel algorithms improve the quality of the solutions when increasing the number of cores. Summing up, the performance is improved by 2.99% (RC and IS 4 cores); 3.11% (RC and 8 cores), 3.25% (IS 8 cores); and 3.29% (RC 12 cores), 3.27% (IS 12 cores). Figures 3(a) and 3(b) also depict the power of the proposed parallel algorithm where for a typical execution, the top performance of the sequential algorithm with the 10-minute time limit is reached in 13 seconds for *random conflict* (4 cores) and 6.5 seconds for *random conflict* (12 cores) for this particular instance

[1] We observed a similar behaviour in the remaining executions

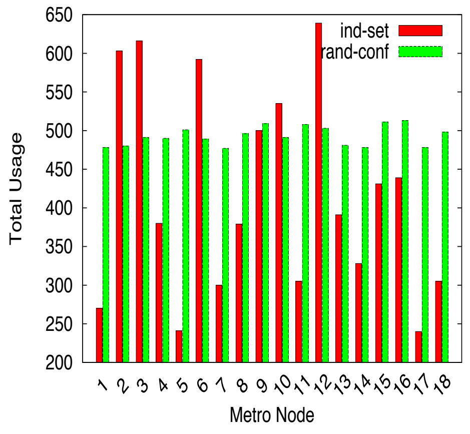
of the Irish dataset. This super-linear speed-up can be explained by the fact that the parallel algorithm applies as many moves as possible in parallel, while the sequential algorithm is improving the quality of different metro nodes (one at time). Although our evaluation has been limited to 12 cores, we would like to remark that the parallelisation degree of the algorithm is limited to the number of metro cores of a given instance (up to 90 metro cores for the UK in our experiments, and it might go up to few hundred metro nodes for countries like Italy, France, and Germany). Moreover, in practice the *fcg* is a sparse graph and therefore we foresee very good speedups up to an important number of cores, e.g., 40 or 50 nodes in the case of UK.



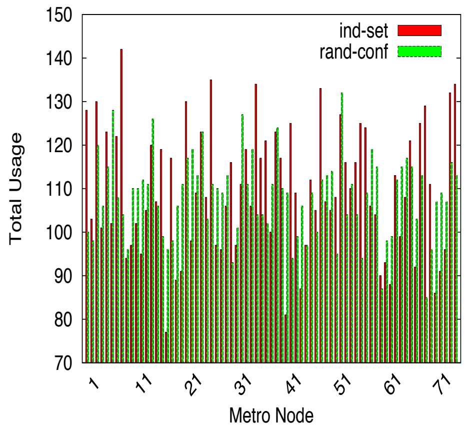
(a) Solution quality (Ireland)



(b) Solution quality (UK)



(c) Selection frequency (Ireland)



(d) Selection frequency (UK)

Fig. 3. Performance summary for a particular execution of an Irish instance (18 Metro Nodes), and a UK instance (75 Metro Nodes).

Finally, we recall that even though after the 10-minute time limit *random conflict* and *independent set* report a similar behaviour, *random conflict* usually performs slightly better than *independent set* up to 100 seconds, and when the degree of parallelism is high (12 cores in this paper). This can be explained in Figures 3(c) and 3(d), where we depict the total number of times the metro nodes are used in the parallel algorithm in a typical execution of the parallel algorithm. As it can be observed, *random conflict* selects metro nodes uniformly. And the nature of *independent set* (see Algorithm 3) bias the selection towards metro nodes with small degree in *fcg*. For instances, the less used metro nodes in the Irish dataset, i.e., metro nodes 5 and 17 in Figure 3(c), have the highest degree in *fcg*, and the more used metro nodes, i.e., metro nodes 2, 3, 6, and 7 in Figure 3(c) have the smallest degree in the connectivity graph *fcg*.

7 Conclusions and Future Work

We have presented an efficient local search algorithm for solving Disjoint Rooted Distance-Constrained Minimum Spanning-Trees problem. We presented two novel move operators along with their complexities and an incremental evaluation of the neighborhood 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 make sense 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, and the UK.

In future we would like to extend DRDCMST with the notion of optional nodes, since this extension is a common requirement in several applications of DRDCMST. 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.

Acknowledgments

This work was supported by DISCUS, and FP7 Grant Agreement 318137.

References

1. Payne, D.B.: FTTP deployment options and economic challenges. In: Proceedings of the 36th European Conference and Exhibition on Optical Communication (ECOC 2009). (2009)
2. Oh, J., Pyo, I., Pedram, M.: Constructing minimal spanning/steiner trees with bounded path length. Integration **22**(1-2) (1997) 137–163

3. Ruthmair, M., Raidl, G.R.: A kruskal-based heuristic for the rooted delay-constrained minimum spanning tree problem. In: Computer Aided Systems Theory-EUROCAST 2009. Springer (2009) 713–720
4. Leitner, M., Ruthmair, M., Raidl, G.R.: Stabilized branch-and-price for the rooted delay-constrained steiner tree problem. In Pahl, J., Reiners, T., Vo, S., eds.: INOC. Volume 6701 of Lecture Notes in Computer Science., Springer (2011) 124–138
5. Ruthmair, M., Raidl, G.R.: Variable neighborhood search and ant colony optimization for the rooted delay-constrained minimum spanning tree problem. In Schaefer, R., Cotta, C., Kolodziej, J., Rudolph, G., eds.: PPSN (2). Volume 6239 of Lecture Notes in Computer Science., Springer (2010) 391–400
6. Khamsi, M.A., Kirk, W.A.: An Introduction to Metric Spaces and Fixed Point Theory. Wiley (2001)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). First edition edn. W. H. Freeman (1979)
8. Hoos, H., Stütze, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann (2005)
9. Hentenryck, P.V., Michel, L.: Constraint-based local search. The MIT Press (2009)
10. Dung, P.Q., Deville, Y., Hentenryck, P.V.: Constraint-based local search for constrained optimum paths problems. In: CPAIOR. (2010) 267–281
11. Baraglia, R., Hidalgo, J.I., Perego, R.: A parallel hybrid heuristic for the tsp. In: EvoWorkshops. (2001) 193–202
12. Crainic, T.G., Gendreau, M.: Cooperative parallel tabu search for capacitated network design. J. Heuristics **8**(6) (2002) 601–627
13. M. Verhoeven, M.S.: Parallel local search for steiner trees in graphs. Annals of Operations Research **90** (1999) 185–202
14. Martins, R., Manquinho, V.M., Lynce, I.: An overview of parallel sat solving. Constraints **17**(3) (2012) 304–347
15. Caniou, Y., Diaz, D., Richoux, F., Codognet, P., Abreu, S.: Performance analysis of parallel constraint-based local search. In: PPOPP. (2012) 337–338
16. Verhoeven, M., Aarts, E.: Parallel local search. Journal of Heuristics **1**(1) (1995) 43–65
17. Shylo, O.V., Middelkoop, T., Pardalos, P.M.: Restart Strategies in Optimization: Parallel and Serial Cases. Parallel Computing **37**(1) (2011) 60–68
18. Arbelaez, A., Codognet, P.: Massivelly Parallel Local Search for SAT. In: ICTAI'12, Athens, Greece, IEEE Computer Society (November 2012) 57–64
19. 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 (June 2013)
20. Roli, A.: Criticality and Parallelism in Structured SAT Instances. In Hentenryck, P.V., ed.: CP'02. Volume 2470 of LNCS., Ithaca, NY, USA, Springer (September 2002) 714–719
21. Michel, L., See, A., Van Hentenryck, P.: Parallel and distribited local search in comet. Computers and Operations Research **36** (2009) 2357–2375
22. Arbelaez, A., Codognet, P.: A gpu implementation of parallel constraint-based local search. In: PDP'94, Turin, Italy (Febrary 2014)