# Using Local Search for Traffic Engineering in Switched Ethernet Networks

Ho Trong Viet, Olivier Bonaventure, Yves Deville, Pham Quang Dung, and Pierre Francois

Dept CSE, Université catholique de Louvain (UCL), Belgium

**Large switched Ethernet networks are deployed in campus networks, data centers and metropolitan area networks to support various types of services. Congestion is usually handled by deploying more switches and installing higher bandwidth link bundles, although a better use of the existing infrastructure would allow to deal with congestion at lower cost. In this paper, we use constrained-based local search and the COMET language to develop an efficient traffic engineering technique that improves the use of the infrastructure by the spanning tree protocol. We evaluate the performance of our scheme by considering several types of network topologies and traffic matrices. We also compare the performance of our technique with the performance that a routing-based deployment supported by an IP traffic engineering technique would obtain.**

***Index Terms*— Combinatorial Optimization, Local search, Spanning Tree Protocol, Traffic Engineering.**

## I. Introduction

During the last ten years, the Ethernet technology has replaced the other Local Area Network technologies in almost all enterprise and campus networks. Furthermore, many Service Providers deploy Ethernet switches in metropolitan and access networks. Last but not least, data centers are largely based on Ethernet switches. In many of these environments, the required bandwidth is growing quickly and network operators need to find solutions to ensure that their switched network can sustain the traffic demand without having overloaded links.

In Ethernet networks, multiple active paths between switches ensure the service availability in case of link failures. Unfortunately, Ethernet networks can not contain active cycles. To avoid such cycles, Ethernet switches implement the IEEE 802. 1d Spanning Tree Protocol (STP) [11] that reduces the topology of the switched network to a spanning tree. When a network segment is unreachable, the reconfiguration of the spanning tree ensures the recovery of connecting. Consequently, the standby paths are reestablished by activating the ports in Blocking State on some switches [5]. The creation of the spanning tree is based on the least cost (shortest) path from each switch to an elected root switch [11]. In this paper, our objective is to provide a Traffic Engineering (TE) technique that optimizes the link weights configuration in order to guide the STP to select an optimum spanning tree for a given traffic demand matrix.

To optimize the choice of the spanning tree by STP, two main approaches are possible. First, we can optimize the spanning tree by searching the link weight space: try to change the weight of each link and thus the shortest path from each switch to the root switch. The problem of this method is the size of the link weight space and the small impact of each change of link weights on the spanning tree. The second possibility is to search the spanning tree space. This method seems better but the size of the search space is still exponential (i.e. $\binom{m}{n-1}$ with $n$ the number of switches and $m$ the number of links). This is the solution chosen in this paper. Once an optimal spanning tree is found, one has to determine link weights such that STP yields the same spanning tree.

Constraint Programming (CP) [14] and Constraint-Based Local Search (CBLS) [17] are well suited for solving complex combinatorial problems. COMET [17] is an object-oriented language with several innovative modeling and control abstractions for CP and CBLS. In COMET, some classical problems can be modeled in only about a dozen lines of code. We chose a local search approach implemented in COMET for solving this TE problem.

Many TE techniques have been proposed for IP, MPLS and optical networks in the last decade [2]. TE allows to optimize the network by redirecting flows to reduce the congestion in the network. In IP networks, TE is usually performed by tuning the OSPF link weights [1], [4]. In MPLS networks, TE techniques use constrained MPLS Labeled Switched Paths to redirect traffic flows around congested links [6]. In optical networks, TE techniques allocate traffic flows to wavelengths [21]. Several approaches have been advocated for solving the TE problem for switched Ethernet networks with the IEEE 802.1s Multi Spanning Tree Protocol (MSTP) [12]. The construction algorithms in [16], [22] and the admission control algorithm in [23] aim to address the TE problem in Metro Ethernet using MSTP. These three works take into account multi-objectives ensuring the load balancing and avoiding the congestion of the spanning tree links. [7] proposes an optimization model for three different load balancing objectives using the heuristic techniques.

However, to the best of our knowledge, there is no previous work dealing with the optimization on the link weight configurations for Ethernet networks using 802. 1d STP [11]. In this paper, we evaluate our work by the improvement of solution quality in each test and by comparing this solution with the one obtained with the IGP Weight Optimization (IGPWO) in [9]. The goal of this comparison is to see if there is a large distance between our solutions and the ones of IGPWO where the routing is offering of Equal Costs Multi Paths.

The remainder of this paper is organized as follows. We first describe the principle of STP in Section II. Second, we define the problem formulation in Section III. Next, we present our local search algorithm with the techniques for speeding up the search in Section IV. Our evaluation is

Manuscript received January 25, 2010. Corresponding author: Ho Trong Viet (e-mail: trong.ho@uclouvain.be).

presented in Section V with an analysis of the experimental results, and we conclude the paper in Section VI.

## II. Spanning Tree Protocol

The purpose of the Spanning Tree Protocol (STP) [11] is to reduce the topology of a switched network to a tree that spans all switches. To calculate the spanning tree, special messages called *configuration bridge protocol data units* (*configuration BPDUs*) are sent by the switches. Each of these messages contains the following information [20]:

- Root ID: ID of the switch assumed to be the root.
- Transmitting bridge ID: ID of the switch transmitting this configuration message.
- Cost: Cost of the least-cost path to the root from the transmitting switch.

At first, each switch assumes itself to be the root, sets the Root ID to its ID, sets the Cost value to 0 and transmits its *configuration BPDU* on every port. *Configuration BPDUs* are distributed in the network so that each switch receives the *configuration BPDUs* of its neighboring switches. The switch with the smallest ID is elected as the root. Thus, each time a switch receives such a message, it compares the Root ID contained in the message and its current Root ID. If the new Root ID is smaller, it generates its own configuration *BPDU* (Root ID, Transmitting bridge ID, and new Cost to the root) and transmits this *BPDU* on its ports. If there is more than one least-cost path to the root, the switch selects the one which passes via the lowest ID neighbor. The spanning tree is created when all the switches have slected the same Root ID and their cost of the shortest path to root are correctly calculated.

Obviously, the most important parameters that determine the resulting spanning tree are the switch IDs and the link costs. With STP these two parameters can be configured.

## III. Problem formulation

We consider our Ethernet network as an undirected graph *G=(N,E)* where N is the set of switches and E is the set of links between switches. Each link *(s, t)* has a bandwidth *BW[s,t]* (note that *BW[s,t]=BW[t,s]*). When link bundles are used between switches, we consider each bundle as a single link having the bandwidth of the bundle. We call *W* the matrix of link weights (STP cost by default - 802.1d [11]) and *TD* the matrix of traffic demands, *TD[s,t]* represents the traffic that switch *s* sends to switch *t*. We call *STP*(*G,W*) the spanning tree obtained by the Spanning Tree Protocol on graph *G,* with link weights *W*. The traditional Ethernet switching problem is defined as follows: for all pairs of nodes *(s,t)* so that *TD[s,t]>0*, distribute the traffic demand over the unique path from *s* to *t* in *STP*(*G,W*).

We call *L* the matrix of load on each link *(s,t)* in the spanning tree: $\mathbf{L[s,t]=\sum\ flow\ over\ (s,t)}$. For the computation of *L[s,t]*, the flow traffic is directed. This means that on a link *(s,t)*, *L[s,t]* is different from *L[t,s]*.

The utilization of a link *(s,t)* is the ratio between its load and its bandwidth $U[s,t]=\frac{\mathrm{L[s,t]}}{\mathrm{BW[s,t]}}$. Like the computation of link load, the link utilization is directed. The link load is kept within the capacity if $\mathbf{U[s,t]\le 1}$ and $\mathbf{U[t,s]\le 1}$. If the one of these two values goes above 100%, the link *(s,t)* is overloaded. These definitions of link load and utilization are equivalent to those used in [4].

In this work, our objective is to find an optimal configuration of link costs $W^*$ minimizing the maximal utilization $U_{max}$ :

$\mathbf{U_{max} = max\{max(U[s,t],U[t,s]) \mid (s,t) \in E\}}$

The formulation of this problem is the following:

**Input:** Graph *G=(N,E)*, bandwidth matrix *BW*, traffic demand matrix *TD*

**Output:** Link cost matrix $W^*$ such that *STP*(*G,W*) yields a spanning tree minimizing $U_{max}$

Solving this problem is a very difficult task as the search space is exponential. Exact methods are not appropriate especially for large instances. In this paper, we focus on approximated methods based on local search.

## IV. Spanning Tree Optimization using Local Search

Local Search (LS) is a powerful method for solving computational optimization problems such as the Vertex Cover, Traveling Salesman, or Boolean Satisfiability. The advantage of LS for these problems is its ability to find an intelligent path from a low quality solution to a high quality one in a huge search space. This can be done by iterating a heuristic of exploration to the neighborhood solutions [17]. In this section, we firstly present an overview of our local search algorithm called LSA4STP (Local Search Algorithms for the Spanning Tree Protocol problem). Then, we describe the

**Algorithm 1** Pseudo-code for LSA4STP

```
1:  root = getRoot(G)
2:  SP = initSpanningTree(root, G)
3:  U*max = getMaxUtilization(SP, BW, TD)
4:  while (time_exec < time_window) do
5:    /* link to (so, to) ∈ SP to be removed */
      selectRemove (so, to, SP)
6:    /* link to (sI, tI) ∉ SP to be added */
      selectAdd ((sI, tI), (so, to), SP)
7:    SP = replaceEdge(SP, so, to, sI, tI)
8:    Umax = getMaxUtilization(SP, BW, TD)
9:    if Umax < U*max then
10:     U*max = Umax
11:     SP* = SP
12:   end if
13: end while
```

algorithms and the techniques allowing to break the symmetries, to define the neighborhood structure, to incrementally compute the link loads and to guide the search. Furthermore, we show how to speed up queries on the spanning tree and to generate the link cost matrix ensuring that the STP computes the intended spanning tree with low complexity.

Our local search algorithm LSA4STP aims to find the best (possible) solution in the spanning tree space. Algorithm 1 provides the pseudo-code of our local search algorithm.

The method *initSpanningTree(root, G)* (Line 2) returns an initial spanning tree as the initial solution for the local search algorithm. In our implementation, we simply simulate the STP

(using some initial *W*) to compute this initial solution. We use the time window as the termination criteria (Line 4). The choice of time window depends on the test size (number of nodes and number of links).

At each search step, we try to replace an edge in the current spanning tree to minimize $U_{max}$.

### A. Root Selection

Symmetry breaking [10] is a well-known technique in Constraint Satisfaction Problems (CSPs) to speed up the search. As described in Section II, to determine a spanning tree, the STP needs an elected root switch and a link cost matrix. For a spanning tree *SP* containing *(n-1)* edges, if the root is not fixed, there are *n* instances of *SP*: $SP_1$, $SP_2$,.., $SP_n$ with *n* different roots (*n* the number of nodes). This means that our search space will be expanded to $n.\binom{m}{n-1}$ spanning trees (with *m* the number of links) if we must perform the search for all the different possible roots.

To eliminate all the symmetries in this problem, we fix a unique root during the whole search process. This has no impact on the single path between any pair of nodes in the spanning tree and has no impact on $U_{max}$. Although the root determination does not change solution, it can change the choice of the neighborhood solution in each search step. In addition, the root influences also the balance of the tree. Network operators normally configure the switches with the highest capacity (ports x bandwidth) as the root of their spanning trees. In our algorithm, we a priori select the node with maximal sum of associated link capacities (bandwidth) as the root. The method *getRoot()* in Line 1 of Algorithm 1 returns such a root.

### B. Neighborhood Formulation

In the search process, we try to move in the spanning tree space to find the solution with the smallest $U_{max}$. In this local search, two spanning trees are considered as neighbors if they differ by one unique edge. The principle of our algorithms is that in each search step, we select an edge to be removed (Line 5 - Algorithm 1) and create a new spanning tree by adding another edge (Line 6 - Algorithm 1). In local search, the definition of the neighborhood is always a key design decision.

#### 1) Removing an edge

In order to reduce the maximal link utilization $U_{max}$, a natural question is: how to lighten the load of the most congested link in the spanning tree. An extreme solution is to replace this link with another one. To determine an edge to be removed, LSA4STP accomplishes the following steps.

First, we find the most congested oriented link *(s,t)* ($U_{max}$=*U[s,t]*) of *SP* that is not in Tabu list (see Section IV.D). Second, from *(s,t)*, we obtain a set *SR* of candidate edges to be removed that contains *(s,t)* and all the edges belonging to the subtree dominated by *s*, as illustrated in Fig 1. Because *(s,t)* is the most congested link, we can assume that this congestion is caused by the traffic coming from the subtree dominated by *s*. Third, we denote *TR* the tree with root *t* containing the edges in *SR*. If we go from the leaves to root *t*, the more we climb the more the traffic increases. Our heuristic

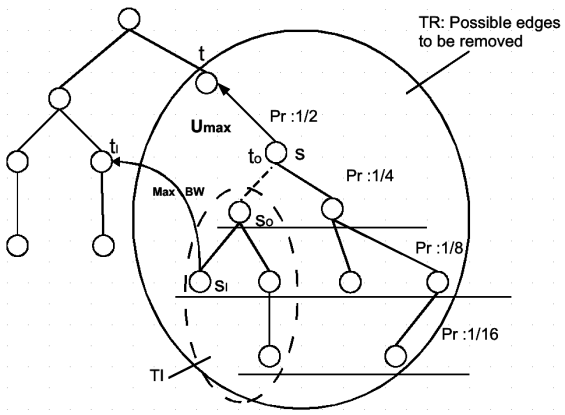


Fig. 1. Selection of the edge to be removed and edge to be added

in this step is to assign to each edge in *SR* a probability to be selected based on its level in *TR*, as illustrated in Figure 1. The edges closer to the root have a higher probability to be removed. The sum of the probabilities associated to the edges at level *i* is $1/2^{i}$, except at the last level *d*, where it is $1/2^{d-1}$ to make the probabilities sum to 1. Last, an edge ($s_O$,$t_O$) is selected based on these probabilities. This selection is performed in COMET by using selector *selectPr*. The idea behind here is to have an effective choice strategy: balancing intensification (greedy search for a solution) and diversification (consider unexplored neighborhood).

#### 2) Adding an edge

After having removed ($s_O$,$t_O$) from *SP*, we obtain two separate trees. We de note *TI* the isolated subtree (unconnected to the root). These two trees must be reconnected with an edge. Our objective is to have more bandwidth and a less congested solution. We consider two criterions for choosing an edge to be added to form the new spanning tree.

First, we define a set of edges *SA* that contains all the edges that join *SP\TR* and *TI*. Second, we select *k* edges having the highest bandwidth from *SA* to form into the set $S_{maxBW}$. Next, we compute the resulting $U_{max}$ when adding each edge of $S_{maxBW}$. Last, the edge ($s_I$, $t_I$) in $S_{maxBW}$ offering the minimal value of $U_{max}$ is selected, as illustrated in Fig 1. For this step, we evaluate only *k* edges because it is more cost-effective than evaluating all the edges in *SA*. In our experiments, *(k=10)* gave the good results.

### C. Speeding up Link Load Computation

For the local search algorithm, it is important to visit as many points in the search space as possible. In this problem, the computation of link loads in each search step is a complex task. To compute the link loads, we must recompute all pairs paths in spanning tree and then the loads over these paths. These computations have a time complexity of $O(n^2 log(n))$.

However, we can show that each time we replace an edge ($s_O$,$t_O$) by another edge ($s_I$, $t_I$), the load changes only on the links on the cycle *C* created by adding ($s_I$, $t_I$) (see Figure 2). Thus, we can avoid recomputing the all pairs paths

recomputation. We compute the link loads as follows: first, assign 0 to the load of $(s_O,t_O)$ because it is removed from the tree. Next, for each pair of nodes *(u,v)* with *TD[u,v]>0*, we verify whether the path from *u* to *v* passes the edges of *C* or not. Then, we recompute the loads on the cycle *C*. We can then benefit from the computations that we performed on the spanning tree before replacing $(s_O,t_O)$ by $(s_I, t_I)$ to compute the cycle *C* and its loads. The size (number of edges) of *C* depends on the network topology. The size of *C* is seldom larger than ten edges in our tests.

In our experiments, the speeding up computation gains about 20% of execution time on each search iteration with the tests where the number of vertices is greater or equal to 100 nodes.

### D. Tabu List

We use tabu search [8] - a heuristic offering a diversified search in each of its search steps. The goal of tabu search is to prevent the search from visiting the same points in the search space. In our problem, a solution is represented by a spanning tree. We can not store and mark all the visited spanning trees

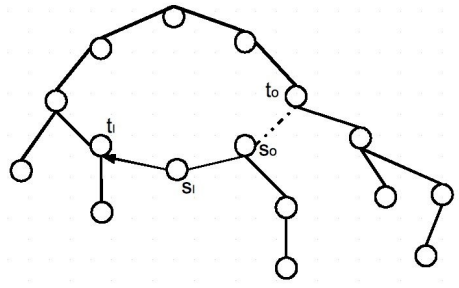


Fig. 2. Loads computed incrementally

because of the expensive space complexity and time complexity to detect if a spanning tree has already been encountered. However, we implement tabu by forbidding the repetitive replacement of a couple of edges in successive iterations.

In LSA4STP, the considered (max congested) edge, the removed edge and the added edge are inserted into the tabu list at each search iteration. We freeze these edges for the next *x* search iterations.

In our experiments, we obtained the good results by setting *(x=10)*.

### E. Spanning Tree Query

In our algorithm, we manipulate a dynamic spanning tree with two kinds of actions at each step of the local search algorithm: (1) update the tree (i.e., edge replacement) and (2) query the tree (i.e., nearest common ancestor of a two vertices, the father of a given vertex, etc.). Queries are usually performed many times in the neighborhood exploration phase. For instance, in order to determine whether or not an edge can be used to reconnect two disconnected sub-trees (by removing an edge of the current tree), we must check whether or not a given vertex belongs to a given subtree. This can be done by querying the nearest common ancestor of two vertices. To simplify the design of our algorithm and to increase its efficiency, we use the VarSpanningTree abstraction of the LS(Graph & Tree) framework [18] representing dynamic tree of a given graph. LS(Graph & Tree) is a local search framework (an extension of COMET [17]) which simplifies the modeling of Constraint Satisfaction Optimization Problems on graphs and trees. Using this framework, many complex computations on trees are modeled and abstracted as a simple query. Last, by using the incremental data structures (auto-update after each change of the tree), all queries on spanning tree mentioned above can be performed in time O(1) and the update action is performed in O(*n*) where *n* is the number of vertices of the given network.

### F. Link Cost Generation

As defined in Section III, our objective is to find an optimal configuration of the link costs $W^*$ such that $STP(G,W)$ yields a spanning tree minimizing $U_{max}$. From the spanning tree obtained by LSA4STP, we generate the cost matrix *W* by assigning a unit cost to all the edges in the spanning tree and by assigning a cost of *n* (number of nodes) to all the other edges in graph. After this assignment, we can see that the cost of the longest possible path between a pair of nodes in spanning tree is *n-1* (passes *n-1* edges) while the cost of the shortest path between any pair of nodes with out using of spanning tree edges is *n* (passes one edge). Consequently, the 802.1d protocol [11] will produce the intended spanning tree.

## V. Experiments and Results

We present in this section the method for generating network topologies and traffic demand matrices for our tests are described. Next, we analyze the obtained results by evaluating the improvement of the link utilization in each test

**Algorithm 2** Pseudo-code for generating Tree Topology

```
1:  root = 1
2:  in_tree = {root}
3:  considered = ϕ
4:  while (#in_tree<n) do
5:    select u ∈ in_tree && u ∉ considered
6:    select num_branch ∈ [min..max]
7:    for all i ∈ [1.. num_branch] do
8:      if #in_tree<n then
9:        select v ∈ [1..n] && v ∉ in_tree
10:       createEdge(u, v)
11:       in_tree = in_tree + {v}
12:     end if
13:   end for
14:   considered = considered + {u}
15: end while
```

and by comparing this solution with the one obtained with IGPWO in [9] assuming that switches would be replaced by routers.

### A. Data Generation

For testing LSA4STP, we generated three types of generic topologies: Grid, Cube, Expanded Tree and use two data center topologies: the PortLand Data Center Network [19] and

the Fat Tree [15]. The traffic demand matrices were generated by using a uniform distribution for *num_des* (configurable) pairs of switches in the network.

*1) Generic topologies*

We present in this section three generic topologies used in our tests. The generation of these topologies has been done as follows:

***Grid*** In a grid topology, we consider each switch as a node on the grid. We can see that in this grid topology, a node has at most four edges. The size of the square grid is $x^2$ with $x$ being the number of nodes on a line. To generate a grid topology with $n$ switches, we choose the smallest $x$ such that $x^2 \geq n$. Among the $x^2$ nodes square grid, we number the nodes increasing from left to right and from top to bottom. Our grid is the part of the square grid containing $n$ nodes from node 1 to

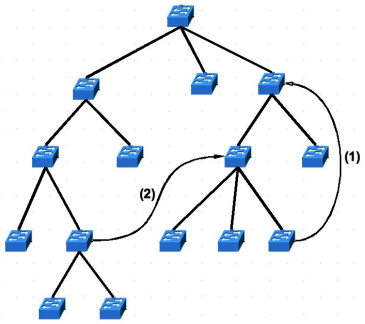


Fig. 3. Expanded tree

node $n$.

***Cube*** We can consider a cube as a composition of $x$ square grids, $x$ being the number of nodes on a line. As for the generation of the grid topology, we choose the smallest $x$ so that $x^3 \geq n$. The cube network is the part containing the $n$ first nodes from the cube.

***Expanded tree*** Tree or hierarchical network is a topology where there is no cycle and the nodes are organized by levels. The generation of the tree topology is described in Algorithm 2. First, we fix the node 1 as the root of tree (Line 1). At each next step, we randomly select a node $u$ that has not yet been considered in the current tree (Line 5). The variable *num_branch* in Line 6 contains the number of child nodes of $u$. The value of *num_branch* is an arbitrary number in interval [*min..max*]. We set (*min=2*) and (*max=6*) to generate our tree topologies. Next, *num_branch* free nodes are selected randomly to be inserted into the tree by creating *num_branch* edges between $u$ and these free nodes (Line 7 to Line 13). The node $u$ is marked as *considered* so that we do not consider it any more in the next steps (Line 14). This step is iterated until all the nodes from 1 to $n$ have been inserted into the tree (Line 4).

The tree obtained by Algorithm 2 is a spanning tree. We add to this tree two types of edges ensuring that we obtain a *biconnected tree* (see Figure 3). The advantage of a *biconnected tree* is: *if any edge is removed, the tree remains connected.* This property ensures that STP has always an alternative solution in case of link failures. In Figure 3, the edges of type (1) connect a leaf with a higher level node while the edges of type (2) connect a non-leaf node (except the root) with a same or lower level node of a different branch. For each tree, (*n-1*) new edges are added to create the *biconnected tree*.

To simulate networks in which a switch has many ports, we define a ratio $r$ ensuring that each node in the tree is connected to at least $r$ edges. In each test, from the generated *biconnected tree*, we create three more trees with ratio $r_{15}=n/15$, $r_{10}=n/10$ and $r_5=n/5$ (where $n$ is the number of nodes).

***Link bandwidths and link weights generation***

For the topologies above, we use two types of link bandwidth: Fast Ethernet 100 Mb/s and Gigabit Ethernet 1Gb/s. In our tests, 80% of the links are Fast Ethernet links and 20% of the links are Gigabit Ethernet links.

The initial link cost matrix $W$ is generated based on the link bandwidths configuration $BW$. According to 802.1d [11], the default link cost for Fast Ethernet is 19 and 4 for Gigabit Ethernet.

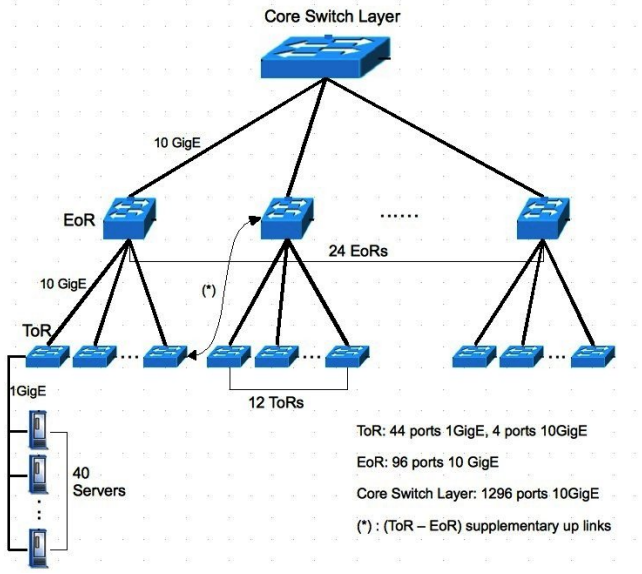


Fig. 4. PortLand Data Center Network

*2) Portland Data Center Network*

Figure 4 depicts the PortLand Data Center Network proposed in [19] consisting of 24 rows. There are 12 racks in each row. Each rack contains 40 machines interconnected by a ToR (top of rack) switch. Each ToR switch has 48 GigE ports: 44 ports of 1 GigE and 4 ports of 10 GigE. Each row has one EoR (end of row) switch containing 96 ports of 10 GigE. Each EoR switch connects to 12 ToR switches via one of the four ports of 10 GigE of the ToR. 24 EoR switches connect to a Core Switch Layer consisting of 1296 ports of 10 GigE. So we have 313 switches in a PortLand topology.

This PortLand design creates a 3-level spanning tree topology. As for the trees in the previous section, we do the same steps to obtain a *biconnected tree*. However, for the PortLand topology, we must consider the available ports on ToR, EoR and Core Switch Layer when adding the edges.

Because PortLand tree has only 3 levels, we aggregate the two types of edges (1) and (2) in Figure 3 as the edges (*) between ToR and EoR as depicted in Figure 4. Each ToR has 4 ports of 10 GigE, one of them is connected to an EoR, so we have 3 available ports of 10 GigE for each ToR. Each EoR has 96 ports of 10 GigE, 12 of these ports are connected to 12 ToRs, 1 another port is used to connect to the Core Switch Layer, and so we have 83 free ports of 10 GigE for each EoR. Core Switch Layer has 1296 ports of 10 GigE, 24 ports have connected to 24 EoRs, and so it has 1272 free ports of 10 GigE. The generated *biconnected trees* of PortLand are considered in our tests.

*3) Fat Tree*

Fat Tree described in [15] is another topology for Data Center Network. It is called ***Fat*** Tree because it is not a spanning tree like PortLand. All the ports on each switch are used to connect to the hosts or to the other switches. The Fat Tree topology is constructed as follows.

All the switches in Fat Tree have $k$ ports. There are $k$ pods. Each pod contains $k$ switches divided into 2 layers: Aggregation and Lower Level. Each layer consists of $k/2$ switches. Each switch in Lower Level connects to $k/2$ hosts and the remaining $k/2$ ports connect to $k/2$ switches of the Aggregation layer of the same pod. There are $(k/2)^2$ switches in the Core layer of a Fat Tree. Each Core switch connects to one switch in the Aggregation layer of all the $k$ pods. We number the switches in the Aggregation layer of each pod from 1 to $k/2$. We split $(k/2)^2$ switches in the Core layer into $k/2$ portions, each containing $k/2$ switches. We number the switches in each of these portions from 1 to $k/2$. The $i^{th}$ switch in each portion of the Core layer is connected to the $i^{th}$ switch in the Aggregation layer of all the $k$ pods. In our tests, we set $k=16$, so we have 320 switches for each Fat Tree topology.

***Link bandwidths and link weights generation***

Like PortLand topologies, all the links of Fat Tree in our tests are 10 GigE and according to 802.1d [11], their link costs are 1.

*4) Traffic Demand Matrix*

Several authors have recently analyzed the traffic matrices that are found in real datacenters [3], [13]. Unfortunately, these datasets are private and there are no methods that can be used to generate traffic demands that are representative of datacenter networks. For this reason, we wrote a simple traffic matrix generator that allows to test several types of traffic matrices. We first start with a uniform traffic matrix where all switches send traffic to all other switches in the network. Then, we generate traffic matrices that are less and less uniform by considering that most of the traffic is sent to a subset of the switches. These non-uniform matrices could correspond to storage servers or routers that often sink a large fraction of the traffic in datacenters.

**ALGORITHM 3** PSEUDO-CODE GENERATING THE TRAFFIC DEMAND MATRIX

```
1:  destinations = getDestinationsSet(num_des);
2:  for all i ∈ [1.. n] && j ∈ destinations do
3:    if i ≠ j then
4:      td[i,j] = getUniformTD(minTD, maxTD, sumTD);
5:    end if
6:  end for
```

The pseudo-code in Algorithm 3 is used to generate the traffic demand matrix for all our tests. We consider that there is a subset of the switches that receive most traffic. In practice, these switches would be the ones attached to routers in data centers or the ones attached to high end servers in Campus networks. The method *getDestinationsSet*(*num_des*) in Line 1 returns the set of destinations for which all switches send the traffic to. From the input parameter *num_des*, this method generates the set of destinations by selecting randomly *num_des* nodes out of $n$ (number of nodes) to insert into *destinations*. In our experiments, we generate for each network topology 5 traffic demand matrices: all switches receive traffic – uniform matrix (*num_des=n*), 50% of switches receive traffic (*num_des=n/2*), 20% of switches receive traffic (*num_des=n/5*), 10% of switches receive traffic (*num_des=n/10*) and 5% of switches receive traffic (*num_des=n/20*). For each network topology, we fix a sum of traffic demand (*sumTD* in Table 1) for all of these 5 traffic demand matrices. This *sumTD* depends on the network type, network size, number of links and link bandwidths (see Table 1). For each pair of switches, the method *getUniformTD* in Line 4 generates a traffic demand in the interval *[minTD..maxTD]*. The *minTD* and *maxTD* values are computed based on *sumTD*.

TABLE I
DATA GENERATION AND TIME WINDOW FOR LSA4STP

| No | Topo. Type | Num. Nodes | sumTD(Mb) | Time Window (s) |
|---|---|---|---|---|
| 1 | Grid | 50 | 400 | 300 |
| 2 | Cube | 50 | 400 | 300 |
| 3 | Expanded Tree 1 | 50 | 1000 | 300 |
| 4 | Expanded Tree 2 | 100 | 1500 | 600 |
| 5 | Expanded Tree 3 | 200 | 2600 | 1200 |
| 6 | PortLand | 313 | 120000 | 1800 |
| 7 | Fat Tree | 320 | 56000 | 1800 |

## B. Experiments

We generated the 7 topologies as described in Table I. Each topology has 5 traffic demand matrices as described in Section V.A.4. For each Expanded Tree, we have four topologies (*biconnected tree*, $r_{15}=n/15$, $r_{10}=n/10$ and $r_5=n/5$) with the same traffic demand matrix. The input of each test is one network topology Graph $G=(N,E)$, one bandwidth matrix $BW$, and traffic demand matrix $TD$. We generated 50 tests for Cube (10 topologies x 5 traffic demand matrices), 50 tests for Grid (10 topos x 5 tdms), 200 tests for each of Expanded Tree from 1 to 3 (40 topos (10 for each of *biconnected tree*, $r_{15}=n/15$, $r_{10}=n/10$ and $r_5=n/5$) x 5tdms), 50 tests for Fat Tree (10 topos x 5 tdms), and 50 tests for PortLand (10 topos (*biconnected tree*) x 5 tdms). So we have 800 tests. These tests are available online in [26]. The time window for running LSA4STP for each topology type is described in Table I.

## C. Evaluation

We consider the improvement of the maximal utilization $U_{max}$ as the criterion to evaluate the performance of LSA4STP. We also measure $U_{max}$ obtained with the default weights by

STP 802.1d.

We present in Figure 5 our results obtained for the 2 generic networks: Cube, Grid as well as Fat Tree and PortLand. For these 4 networks, LSA4STP gives the best results for the Cube since it reduces on average the value of $U_{max}$ up to 50% for all the 5 types of traffic demand matrices. For the Grid, Fat Tree and PortLand $U_{max}$ is also reduced to about 60%-80%. LSA4STP works better for the Cube, Grid than for the Fat Tree, PortLand because the link bandwidths in the Fat Tree, PortLand are homogeneous (all 10 Gb/s) while the ones in the Cube, Grid are not (100 Mb/s and 1 Gb/s). But the reason is not the difference of the sum of link bandwidths between the

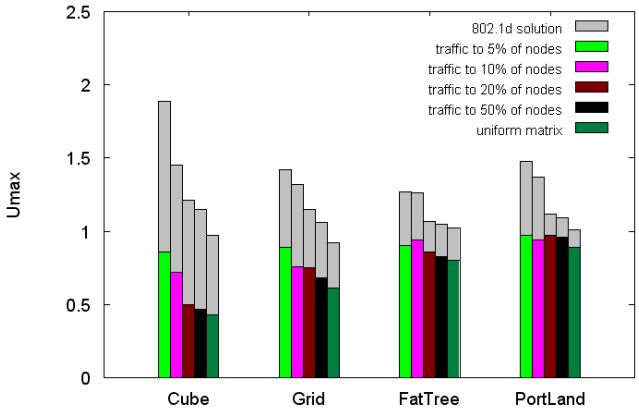


Fig. 5. Result for Cube, Grid, Fat Tree & PortLand

spanning trees. In our tests, this sum of the initial (802.1d) and best $U_{max}$ (LSA4STP) one is almost the same. However, for the Cube and Grid, in the 802.1d solutions the most congested links are the links of 100 Mb/s while the 1 Gb/s ones are not efficiently used as in the solutions obtained with LSA4STP.

Our local search algorithm is especially efficient for the

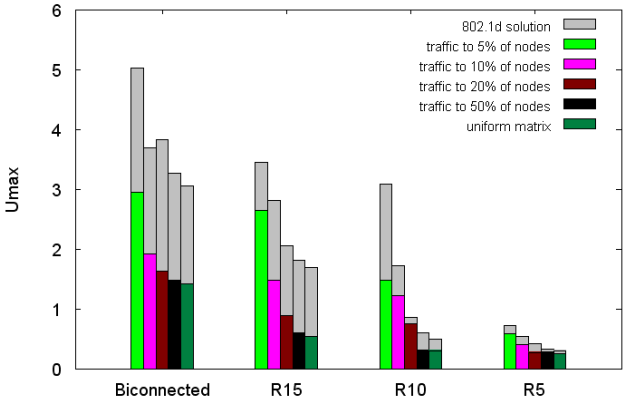


Fig. 6. Result for Expanded Tree 100

tests of Expanded Tree in which high quality solutions exist in a large search space. Figure 6 depicts the results of LSA4STP with the tests of Expanded Tree 100 in which the value of $U_{max}$ is dropped to about 49% for the *Biconnected Tree*, 47% for $r_{15}$, 64% for $r_{10}$ and 78% for $r_5$. The reason is always the efficient use of 1Gb/s links. With the ratio $r_5$, the average number of 1 Gb/s links is 97 out of 99 links of spanning tree

TABLE II
AVERAGE TIME FOR LSA4STP FINDING THE BEST SOLUTION (IN S)

| **Topo. Type** | #des=n | #des=n/2 | #des=n/5 | #des=n/10 | #des=n/20 |
|---|---|---|---|---|---|
| Grid | 192 | 188 | 170 | 185 | 124 |
| Cube | 91 | 165 | 130 | 130 | 206 |
| Exp.Tree 1 | 108 | 103 | 130 | 128 | 143 |
| Exp.Tree 2 | 222 | 224 | 211 | 306 | 271 |
| Exp.Tree 3 | 251 | 232 | 298 | 321 | 258 |
| PortLand | 350 | 339 | 418 | 427 | 412 |
| Fat Tree | 409 | 392 | 438 | 385 | 395 |

(almost homogeneous of link bandwidths). This number of *Biconnected Tree* is 32, of $r_{15}$ is 53, of $r_{10}$ is 76. That explains why with the ratio $r_5$, LSA4STP has the similar results as Fat Tree and PortLand.

In both of Figure 5 and Figure 6, we can see the same decreasing order of $U_{max}$ for each test from left to right. With the same topology and the same sum of traffic demand (*sumTD*), each traffic demand matrix gives a different $U_{max}$ (obtained with 802. 1d or LSA4STP). Higher $U_{max}$ values are observed with the more biased traffic demand matrices. Because for all these 5 traffic demand matrices, all switches send traffic to a number of switches (*num_des*). So the more switches receive traffic the more balancing traffic is

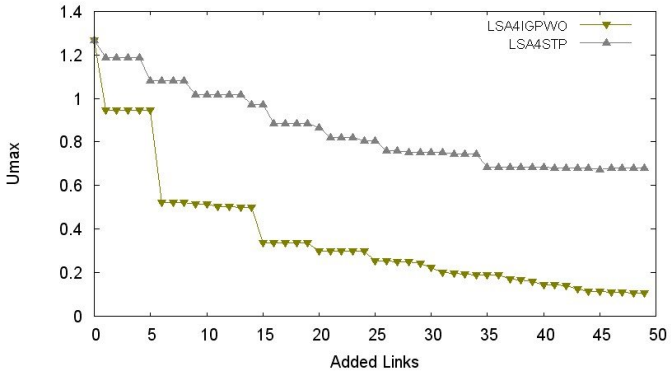


Fig. 7. LSA4STP vs LSA4IGPWO

distributed to the links. Obviously, the uniform traffic demand matrix always gives the lowest $U_{max}$.

We describe in Table II the average time for LSA4STP to find the best solution for each test of the 7 topologies. This time for the topologies of: 50 nodes (Cube, Grid, Expanded Tree 1) is about 2 minutes, 100 nodes (Expanded Tree 2) is ~ 4 minutes, 200 nodes (Expanded Tree 3) is ~ 5 minutes and 313 (PortLand) and 320 (Fat Tree) nodes are ~ 6 minutes. We can state that LSA4STP works well for the large scale tests when the time complexity of link load computations in each search iteration is collapsed.

### *D. Comparison with IGP Weight Optimization*

Several IGP Weight Optimization (IGPWO) techniques have been proposed for IP networks [4], [9], [1], [2]. In this section we compare the performance of LSA4STP to a local search algorithm (LSA4IGPWO) in the COMET language [9]. IGP weight optimization is not applicable to existing Ethernet networks as it only applies to IP routers. Compared to Ethernet

switches, IP routers have the advantage of being able to send packets over all links in the network while the STP disables a subset of the links. Given the price difference between IP routers and switches, large datacenters will not replace their switches with IP routers. However, there is ongoing work within the IETF to develop standards to allow next-generation switches to use the IS-IS routing protocol instead of the 802.1d protocol [24], [25]. This solution requires more powerful switches and it can be expected that it will only be supported on new high-end switches initially.

To compare IGPWO with LSA4STP, we performed an experiment with the Expanded Tree containing 50 nodes and the uniform traffic demand matrix (*num_des=n*). The time windows for running IGPWO and LSA4STP are the same (300s). To evaluate the impact of the number of alternate paths in the topology, we varied the number of additional links in the Expanded Tree. Figure 7 shows on the x axis the number of links that were added to the Tree (see section V.A.1) and on the y axis the maximum utilization for IGPWO and LS4STP. When there are less than 6 links, LS4STP is within 20% of the solution obtained by IGPWO. When the number of additional links in the Expanded Tree grows, the distance between LSA4STP and IGPWO grows as well. This is normal since LSA4STP uses only a fraction of the links while IGPWO is able to send traffic over all links. For example, regarding Figure 7, at 25 added links, to obtain an $U_{max} \sim 0.3$, IGPWO must use all the 74 links. So when there are link failures, this solution given by IGPWO has no reserved link to ensure the service availability while the number of reserved links of the one obtained with LSA4STP is 25.

## VI. Conclusion

In this paper, we have proposed a new TE technique based on local search that finds the best spanning tree that minimizes congestion for a given traffic matrix in Ethernet network.

Our choice of directly optimizing spanning trees instead of link weights reduces the size of the search space. We have proposed an efficient technique to recompute the link loads at each search iteration that can avoid the all pairs paths computation.

Our local search heuristic has been implemented in the Comet language and our simulations show promising results.

Our further work is to extend our framework to support multiple traffic matrices and multiple VLANs.

## VII. Acknowledgement

Ho Trong Viet is supported by the FRIA (Fonds pour la formation à la Recherche dans l'Industrie et dans l'Agriculture, Belgium). Pierre Francois is supported by the FRNS (Fonds National de la Recherche Scientifique, Belgium).

## References

[1] A. Sridharan, R. Guerin and C. Diot, "Achieving Near-Optimal Traffic Engineering Solutions for Current OSPF/IS-IS Networks", IEEE/ACM Transaction on Networking, Jan, 2005.

[2] Antonio Nucci, and Konstantina Papagiannaki, "Design, Measurement and Management of Large-Scale IP Networks: Bridging the Gap Between Theory and Practice", Cambridge University Press, 2009.

[3] Benson, T., Anand, A., Akella, A., and Zhang, M, "Understanding data center traffic characteristics", SIGCOMM Comput. Commun. 2010.

[4] Bernard Fortz, and Mikkel Thorup, "Internet Traffic Engineering By Optimizing OSPF Weights, " IEEE INFOCOM, 2000.

[5] Cisco Systems, "Understanding Spanning Tree Protocol Appendix C," Cisco Systems 1989-1999pp, C-1 through C-12.

[6] D.O. Awduche, "MPLS and Traffic engineering in IP Networks", IEEE Communications Magazine, Dec. 1999, pp. 42-47.

[7] Dorabella Santos, et al., "Traffic Engineering of Multiple Spanning Tree Routing Networks: the Load Balancing Case", Next Generation Internet Networks NGI '09, 2009.

[8] Glover, F. and M. Laguna, "Tabu Search", Kluwer Acadenic Publishers, Norwell, MA, 1997.

[9] HO Trong Viet, P. Francois, Y. Deville, and O. Bonaventure, "Implementation of a Traffic Engineering technique that preserves IP fast reroute in COMET," Proceedings of the 2009 AlgoTel, June 16-19, 2009.

[10] I. Gent, and B. Smith, "Symmetry breaking in constraint programming," *Proc. ECAI00*, 2000.

[11] IEEE Standard 802.1D, "Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Common specifications-Part 3: Media Access Control (MAC) Bridges," ,1998.

[12] IEEE Standard 802.1S, "Virtual Bridged Local Area Networks - Amendment 3: Multiple Spanning Trees", 2002

[13] Kandula, S., Sengupta, S., Greenberg, A., Patel, P., and Chaiken, R, "The nature of data center traffic: measurements & analysis", Proceedings of the 9th ACM SIGCOMM Conference on internet Measurement Conference , Chicago, Illinois, USA, November 04 - 06, 2009.

[14] Krysztof R. Apt, "Principles of Constraint Programming", Cambridge University Press, 2003.

[15] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *SIGCOMM* 2008.

[16] M. Padmaraj , et al., "Metro Ethernet traffic engineering based on optimal multiple spanning trees", Wireless and Optical Communications Networks, 2005. WOCN 2005.

[17] P. V. Hentenryck, and L. Michel, "Constraint-based Local Search," MIT Press, 2005.

[18] Pham Quang Dung, Y. Deville, and P. V. Hentenryck, "LS(graph & tree): a local search framework for constraint optimization on graphs and trees," Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), 2009.

[19] R. N. Mysore, and al., "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric," *SIGCOMM* 2009.

[20] Radia Perlman, "Interconnections: Bridges, Routers, Switches, and Internetworking Protocols (2nd Edition)," *Addison-Wesley*, 1999.

[21] Ramaswami, R. and Sivarajan, K. N, "Routing and wavelength assignment in all-optical networks. IEEE/ACM Trans. Netw, 1995.

[22] Wentao Chen, Depeng Jin, and Lieguang Zeng, "Design of Multiple Spanning Trees for Traffic Engineering in Metro Ethernet", International Conference on Communication Technology ICCT '06, 2006.

[23] Xiaoming He, Mingying Zhu, and Qingxin Chu, "Traffic Engineering for Metro Ethernet Based on Multiple Spanning Trees", Conference on Mobile Communications and Learning Technologies, 2006.

[24] R. Perlman, "RBridges: Transparent Routing", Proc. IEEE INFOCOM 2005.

[25] R. Perlman et al., "RBridges: Base Protocol Specification", Internet draft, Jan. 2010.

[26] Ho Trong Viet et al., "Using Local Search for Traffic Engineering in Switched Ethernet Networks", 2010. http://becool.info.ucl.ac.be/page/datas-lsa4stp