UCL
Université catholique de Louvain

Université Catholique de Louvain

École Polytechnique de Louvain

École Polytechnique de Louvain

Local Search for Multicast in Software-Defined Networks

Supervisors:
Bonaventure Olivier
Deville Yves

Reader:
Aubry François

Thesis submitted for the Master's Degree
in Computer Science and Engineering by
Debroux Léonard
Jadin Kevin

Louvain-la-Neuve
Academic year 2013-2014

Abstract

Software-defined networking (`SDN`) provides additional knowledge compared to classical networks. This knowledge can be exploited in centralised algorithms as opposed to the commonly used distributed algorithms such as in `PIM`. The problem of implementing multicast in `SDN` has already been studied in the literature. However, the optimality of multicast trees is often not the main focus of the researches.

In this thesis, we show that the `SDN` approach allows for easily building efficient trees, and we provide a method to further improve them. The problem of computing optimal distribution trees in the context of multicast is known to be NP-complete, we therefore propose a configurable local search algorithm to compute competitive trees along with a Python implementation.

Furthermore, we assess that our algorithm is computationally efficient. It is consequently applicable in a real environment since it needs little time to yield good solutions.

Acknowledgements

We would like to express our deep gratitude to Professor Olivier Bonaventure and Professor Yves Deville, our Master's thesis supervisors, for their continuous presence.

Our regular meetings helped us keep our progress on schedule. The appropriate and sensible critiques they provided us with during the thesis helped us focus on our final objectives and answered every problem we could not solve. Also, their unfailing availability allowed us to keep a steady pace.

Furthermore, we would like to thank them for teaching us to persevere throughout the sometimes discouraging results we met during the development of our approach. Their guidance was an undeniable asset in the realisation of this work. Our thanks go to François Aubry, our reader, for taking the time to review our thesis.

We are also grateful to our school and past teachers for our education. They made us master the concepts and methodologies that we applied in this thesis.

Finally, we wish to thank our parents for their never-ending support, urging and encouragements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multicast is a technique to efficiently send information to several receivers in a network. Software-defined networking is an approach on the way to handle networks by decoupling data and control plane.

In this thesis, we want to address the combination of the two by proposing an efficient way to achieve multicast in software-defined networks.

As an introduction, we first explain those concepts, as well as motivate the choice of this subject. We then present the structure of this document.

1.1 Context and environment

This section is based off the "Computer Networks" courses given at UCL by Prof. O. Bonaventure [7, 8].

First, we describe the network environment in which multicast solutions are developed. Secondly, we introduce what multicast actually is. Finally, we describe what software-defined networking is and what this approach can bring in today's networks regarding multicast.

1.1.1 What is a network?

A computer network is a network of inter-connected devices of various kinds that use protocols for exchanging data on links connecting them. Devices have a specific purpose in the network: some devices are responsible for routing data inside the network, while others provide a set of services to end applications. The devices and the links connecting them can be represented on a graph, called the *network graph*.

NETWORK GRAPH

In a network, sending information from a source to a destination means that the information has to travel through the links and the devices of the network.

The path it follows in the network basically depends on the costs attributed to the links. Those costs are the expression of attributes of the links such as the available bandwidth or the load of the links.

1.1.2 Applications of multicast

Multicast is used when a sender needs to distribute the same data to a set of destination nodes. It therefore suits many practical applications.

One common example of multicast technologies is to distribute television through Internet (IPTV technologies). Multicast is well suited for this purpose as it provides a way to deliver a single stream of data to a set of receivers. As television is made of a single video stream that needs to be transmitted in real time, a multicast approach makes sense.
In this specific application, we understand easily that sending two copies of the same video stream to two neighbours is more costly than simply duplicating a single copy of the video when it arrives in front of their two houses.

Another example of application of multicast is for real time peer-to-peer (P2P) applications, such as conference calls or chat rooms, where each peer information needs to be sent to every other peer. For that purpose, multicast is an essential mean for minimising network utilisation.

A last example would be how stock markets can be updated. Such an update scheme based on multicast has been described in [9].

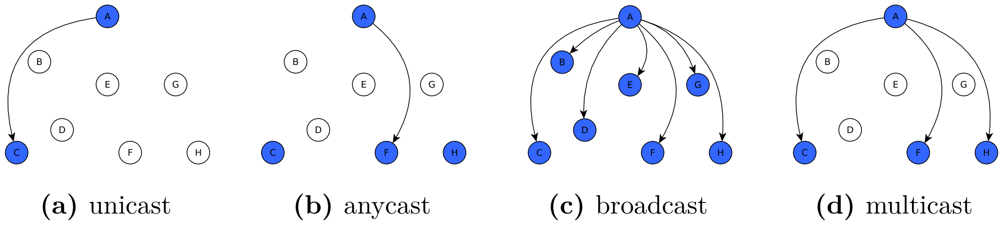
1.1.3 What is multicast?

Multicast is a method for delivering data from a source (or several sources) to a set of receivers – forming a group – without having the need to send an individual copy of the data to each of the receivers. The stream of data to deliver is the same for each receiver.

Several other delivery methods exist. Those are depicted in fig. 1.1. Multicast distinguishes itself from other delivery methods:

unicast : the packet is sent from one source to a single destination
anycast : the packet is sent from one source to at most one of the destinations
broadcast : the packet is sent from one source to everyone
multicast : the packet is sent from one source to a subset of the nodes

When a source wants to send a single stream of data to a set of more than one destination nodes, multicast can reduce network utilisation. A multicast service (fig. 1.1d) can be achieved using other delivery methods:



(a) unicast **(b)** anycast **(c)** broadcast **(d)** multicast

Figure 1.1: Several transmission mechanisms

Using unicast : unicast-based multicast consists in sending the data several times, each time to a different receiver.

Using broadcast : broadcast-based multicast distributes the information to every node independently of their inclusion in the receiver set. Nodes who are not part of the receiver group can then discard unwanted data.

While simple, those ways to achieve multicast also waste a lot of network resources. In the unicast case, each packet addressed to a different receiver must be routed independently and therefore circulates like a normal packet. This waste of resources is even greater when destination nodes are localised in the same area: the same data must be forwarded as many times on the same links as there are destinations.

In the broadcast case, data is sent to all nodes in the network, even if such nodes did not request the data. One can easily see the limitations of this approach, especially if the size of the network grows.

Examples of the unicast and multicast case are given at fig. 1.2a and fig. 1.2b.

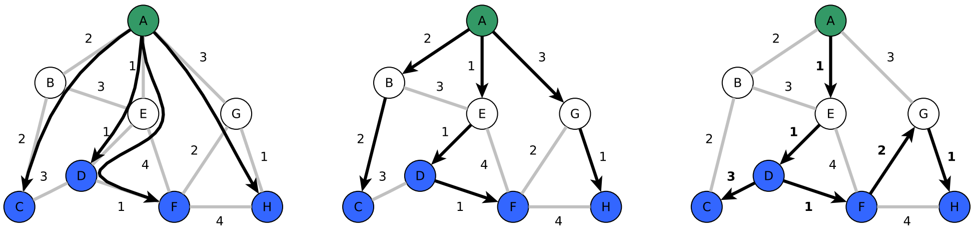
An alternative way for handling multicast in a network is done through *packet duplication*. Following this approach, one could build a *multicast distribution tree* rooted at the source of the stream. Leaves of such a tree would be a subset of the destination nodes, the remaining destination nodes being on the branches of the tree leading to the leaves. This scheme can only be supported with packet duplication. As depicted in fig. 1.2c, which gives an example of such a distribution tree, packets are routed from the source to the destinations. If, from a router's point of view, an incoming packet needs to be forwarded to more than one outgoing link, then the data is duplicated and sent on those links.

PACKET DUPLICATION

MULTICAST DISTRIBUTION TREE

Following this approach, the distribution process can be optimised to limit the total amount of data transferred on the network to a minimum. The data needs to be transmitted only once on each link.

In order to minimise global network utilisation, this distribution tree must be made minimal. This amounts to computing an *optimal* distribution tree. However, as explained in section 1.3.2, finding such an optimal distribution tree is an NP-complete problem.



(a) unicast **(b)** broadcast **(c)** multicast

The cumulated costs of the used edges represents the cost of sending the information to each recipient. In the case of unicast, the cost is 13, in the case of broadcast, the information is send to each node, and the cost is 11. The optimal multicast tree costs 9.

Figure 1.2: Distribution trees. With node **A** as source, and nodes **C**, **D**, **F** and **H** as destinations

In a classical IP networks, where each node makes routing decisions by itself, the computation of this multicast distribution tree must be achieved using external protocols, such as the *Protocol-Independent Multicast* (PIM) protocol family [10]. This protocol family includes several variations to achieve multicast in a standard network. Only one protocol of this family solves the problem described in this thesis: the *Source-Specific Multicast* (`PIM-SSM`) [11], which builds distribution trees rooted at a single source. Indeed, other protocols from this family solve other multicast problems, for example when several sources want to transmit on the same distribution tree. This case will not be directly addressed in this thesis.

However, the classical `PIM-SSM` approach does not achieve optimality, and that for two reasons : the problem is difficult (NP-complete) and nodes in classical networks have to make routing decisions based on their limited knowledge of the network.

As opposed to this classical way of building multicast distribution trees, software-defined networks bring extended knowledge and additional functionalities, and enable the writing of new algorithms for this multicast problem.

1.1.4 What are software-defined networks (`SDN`)?

CONTROL PLANE
DATA PLANE

Software-defined networking describes a set of new methods for network configuration (*control plane*) and handling of data (*data plane*).

The idea is to separate control plane from data plane as much as possible.

CONTROLLER

In that scheme, a *controller* is responsible for handling the control plane in its entirety. Such a controller has a centralised view on the network and knows everything about it : which nodes or links are running and which are down, the degree of utilisation of such links, etc. The controller makes decisions based on this centralised and global view of the network. This scheme is in opposition with current networks as the intelligence is removed from the devices (routers) and centralised in one place.

The physical network responsible for handling and routing packets from their source to their destination(s) is then made of dumber network entities, typically switches, which take decisions based on what the controller decides.

The advantages of the `SDN` approach are numerous. First, as intelligence is removed from the devices composing the physical network, such devices can be made simpler. Their sole purpose is to forward packets, they can then focus on that goal and become simpler machines. Because the overhead is removed from those devices (both in terms of memory and computation mechanisms), they are cheaper to develop and build.

Moreover, the clear separation between control and data plane enables an easier introduction of functionalities in the network, as devices are not bound to the protocols running on them any more. Support for new functionalities only requires the update of the controller, as long as those functionalities are implementable with respect to the specifications which frame the behaviour of switches and their interactions with the controller.

Finally, because the whole state of the network is known by one entity, meaningful decisions can be taken immediately and appropriately as soon as a change occurs in the network (for instance, a failure). This kind of quick response to failure can hardly be achieved in standard networks, where some protocols can take a significant amount of time to converge.

An example of `SDN` are `OpenFlow` networks [12].

What is `OpenFlow`?

The *`OpenFlow` protocol* defines ways to work on a programmable network made of switches. An `OpenFlow` network controller can, by using the `OpenFlow` protocol, issue changes to the programmable switches to define how they should handle and/or forward packets being switched in the network.

OpenFlow protocol

Several versions of the `OpenFlow` specifications exist. They define how the controller communicates with the switches as well as how those switches should forward packets in the network. Each version specifies a set of mandatory features that must be supported by switches, and a set of recommended but not mandatory ones. The range of features has been refined and expanded throughout the release of the successive versions. However, the basics have not evolved.

Stable versions include version `1.0` [13] to the most recent version `1.4` [14], released last October 2013. Commercial `OpenFlow` hardware usually supports `OpenFlow 1.0` only, and rarely a newer version. Virtualised environments, on the other hand, support a wider range of `OpenFlow` versions. As of today, Open vSwitch version `2.3` [15] fully supports `OpenFlow` up to version `1.3` [16], and version `1.4` partially.

IPv6 support was introduced in `OpenFlow` in version `1.2` [17].

An `OpenFlow` switch obeys certain basic rules for forwarding packets to its neighbours. Each packet is processed in a systematic manner. The way an incoming packet is handled is described in switches' *flow tables.* Flow tables are composed of an ordered list of *matching rules* and corresponding *actions.*

FLOW TABLES

MATCHING RULES

ACTIONS

Matching rules define characteristics on which packets can be matched. For example, a certain IP address, or a specific port value.

To each matching rule corresponds one or more actions that must be applied to the packet.

If the packet does not match any rule from the flow tables, it is forwarded to the controller over a secure channel. The controller can then decide what action to perform for this specific kind of packet.

Actions can be of several kinds, among them:

`Drop` : discard the packet is discarded

`Forward` : forward the packet to a local or a virtual port (`ALL`, `CONTROLLER`, etc).

`Modify-Header` : edit a packet header (useful for integration in today's networks)

`...`

1.2 Motivations

There are several reasons to be interested in the application of multicast in an `SDN` environment. The first one is that multicast use is growing every year. To only talk about IPTV, from 2012 to 2013, there was an increase of 21% in subscribers worldwide. In the fall of 2013, the number of subscribers across the globe accounted for 96 millions [18].

Multicast is also used for stock exchange as it requires to send the same information to a large number of receivers at the same time.

As software-defined networking might become a disruptive technology, every feature that is running on the current standard networks will have to be applied to that new paradigm. This kind of technology is now developed by several companies such as Google, Amazon, Microsoft and Facebook. Recently, Google has unveiled *Andromeda* [19], and Cisco revealed *OpFlex* [20], its alternative to `OpenFlow`.

While both multicast and `SDN` are fascinating topics, the `SDN` technology has a lot to bring to the current way of doing multicast. Thanks to the centralised viewpoint we benefit from in the `SDN` approach, we have a global view of the networks and a greater knowledge of the environment.

In standard networks, for example when using PIM, the distribution tree is built based on the sole knowledge of the identity of the source/rendez-vous point. The shape of the active distribution tree is not necessarily known to the routers. With SDN, the whole tree is known by the controller, which can then optimise it in a way that is impossible in current approaches.

In this thesis, we want to study the possibility of computing multicast distribution trees in the context of this global knowledge.

1.3 Problem definition

In this section, we first describe the multicast problem we specifically want to address in this thesis. Some requirements and constraints on its performance are listed.

The problem of building a multicast tree in an environment where the whole graph is known can be expressed as a Steiner tree problem [21]. This is described in section 1.3.2.

Finally, section 1.3.3 describes our choice to use a local search approach to address the multicast problem.

1.3.1 Multicast tree computation problem, its requirements and constraints

As introduced in section 1.1.3, part of solving the multicast problem is finding efficient ways to compute competitive distribution trees in the network. Because of their distributed nature, classical protocols such as the PIM-SSM protocol cannot achieve cost-effective multicast. Those methods often rely on shortest paths to a limited number of known nodes in the graph (source, rendez-vous point).

The emergence of the SDN approach enables us to write new algorithms for building competitive distribution trees, in clear opposition with the distributed protocols currently running in classical IP networks. With the complete knowledge of the topology brought by SDN, we are able to maintain better multicast trees, while still complying with the constraints coming with the on-line aspect of the network environment.
Those requirements are:

- The handling of the arrivals and departures of subscribers (of the multicast group) must be done as quickly as possible. We therefore need an efficient procedure for handling such events;
- The distribution trees should maintain a certain stability. Its structure should not change too often as multicast service might be impacted by such changes.

The impact of changing routes in the network should be limited in order to avoid disturbing the subscribers of the multicast group.

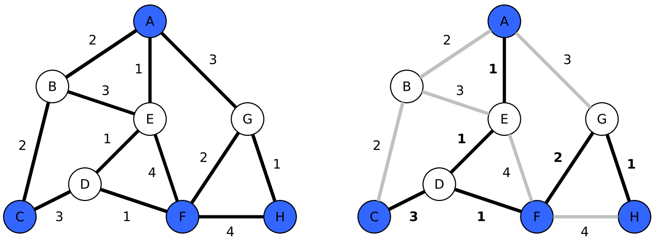
1.3.2 Minimum Steiner tree problem: optimality for multicast

To achieve optimal multicast, we want to be able to compute a minimum cost tree covering the source and every destination. The source and the destinations are vertices of the network graph.

This problem of optimally interconnecting a set of vertices V of a graph is known as the minimum Steiner tree problem [21].

This problem is similar to the minimum spanning tree problem (MST), which is to find the least costly subgraph, representing a tree, that connects all vertices from a graph together. However, the Steiner tree problem is different from the MST problem in the sense that in the latter, the goal is to connect all the vertices together, while the Steiner tree problem aims at interconnecting only the ones in V. Furthermore, a vertex that is not in V can be part of the tree if a route going through it decreases the total weight of the tree.

An example of such a tree in a graph is given on fig. 1.3.



Connecting F and H through G is better than using the edge (F, H) as it decreases the weight of the tree by one unit of cost.

(a) Graph **(b)** Minimum Steiner tree

Figure 1.3: Graph and its Steiner tree for $V = \{A, C, F, H\}$

The other major distinction is that the Steiner tree problem is NP-complete. That means that in any case, computing the perfect multicast distribution tree is currently impossible in polynomial time. It would basically require to go through the 2^m subgraphs that can be built from the graph (with m being the number of edges in the graph). A simple example is given at fig. 1.4.

Researches exist about multicast and the Steiner tree problem, and some of it is presented in section 2.1.

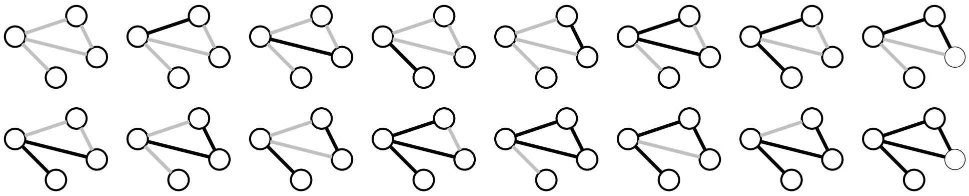


Figure 1.4: 2^m possibilities for a graph of 4 nodes and 4 edges

1.3.3 Local search

This section is based off the “Artificial Intelligence” course given at UCL by Prof. Y. Deville [22].

Because the minimum Steiner tree problem is NP-complete, an exhaustive approach is not possible, the search space being all the possible trees in the graph. For that reason, our algorithm uses a local search approach.

Local search is based on an iterative exploration of the neighbourhood of the current solution. This exploration is done by modifying the current solution in order to try to improve it. Two successive solutions are then close to each other. In our problem, exploration consists in replacing some edges in the multicast tree by some others in order to try to reduce the total cost of the tree.

Local search

Several meta-heuristics and heuristics are included in our local search approach.

A *meta-heuristic* aims at driving the search towards global optimality, in other words, its goal is to avoid getting stuck in locally optimal solutions. We use two meta-heuristics: the simulated annealing and the tabu search.

meta-heuristic

Simulated annealing aims at improving the solution whenever it is possible. When it is not possible to improve it, simulated annealing allows to degrade the solution with a certain probability.
The probability to degrade is bound to a temperature. The higher the temperature, the higher the probability to degrade. There can be several ways of scheduling the temperature during the search. A possible way is making it decrease, so that it becomes less and less likely to degrade over time.

Simulated annealing

Tabu search is the second meta-heuristic used in our method. It prevents the search from visiting a neighbourhood that has already been seen recently. It requires some memory as the search must know where not to go.

Tabu search

On top of meta-heuristics, we defined heuristics to use in our local search approach to drive the search towards optimal solutions (which can be locally-optimal). It does not require any additional memory and is based on the current knowledge of the problem.

For a local search approach to perform efficiently, we must find a good balance between intensification and diversification.

INTENSIFICATION *Intensification* aims at exploring promising local areas of the search space but can then lead to local optima. Intensification is achieved by favouring the good solutions in a solution's neighbourhood during the search.

DIVERSIFICATION *Diversification*, on the other hand, aims at exploring other regions of the search space. The goal is to avoid getting stuck in local optima and is thus sometimes necessary for driving the search towards the global optimal. As it leads to exploring other parts of the search space, convergence to optimality can take more time. Diversification is achieved by bringing randomness to the search and by making probabilistic choices.

1.4 Structure of the thesis

The following steps highlight the structure of our thesis.

First, we describe the state of the art on multicast and software-defined network in chapter 2. We propose a survey of the papers by briefly explaining their content and by highlighting what ideas and results are applicable in our context.

Second, we describe, in an abstract way, the method we developed to build multicast trees. This algorithm is explained in chapter 3. The method is presented by first identifying the main sub-problems which we then split into smaller ones, easier to solve.

After describing our method, we give some details about our implementation. This is done in chapter 4. We motivate the technologies we used, as well as present the different data structures that were created and used. We finally give information about some additional side-algorithms.

In a fourth step, chapter 5 details the experiments we made and the results we obtained. In particular, we compare the results of our method with the one of `PIM-SSM`, and motivate the possibility to use it in a real environment.

Finally, chapter 6 concludes by summing up the content of the thesis. We also discuss the limitations of our approach and describe what future work could be done on the subject.

Chapter 2

State of the art for multicasting in software-defined networks

Solutions exist for tackling the minimum Steiner tree problem, as well as for tackling the multicast problem in an OpenFlow network. The expression of the multicast problem as a Steiner tree problem is explained in section 1.3.2.

We first discuss about the Steiner tree problem and the issue of computing a multicast tree in an online environment, then we discuss about the application of multicast in an OpenFlow network and the solutions that have already been developed.

Of all the solutions proposed in the following papers, none of them is complete. This is due to the NP-completeness of the problem they address.

2.1 Multicast and the Steiner tree problems

The problem of finding a multicast tree can be expressed as the problem of computing a Steiner tree with the multicast source and clients as Steiner nodes. Papers about the Steiner tree problem are thus as relevant as the ones about multicast. In this section, we detail several papers we kept as reference, and explain why they do or do not fit our needs.

We identified several key aspects to the issue we address:

- **Subscription management** Managing the subscribers of the multicast group, which means managing join and leave requests from clients. The active subscribers will be expressed as Steiner vertices in the multicast distribution tree.
- **Complexity** Applying join/leave events as fast as possible, as we want those tree modifications to take effect immediately. This wish is justified by the

online nature of the environment.

- **Optimisation** Trying to keep the multicast distribution trees as good as possible, meaning that the distribution trees should have the lowest possible cost.

In the present section, we discuss five papers that provide leads and ideas for the problem we want to solve. We highlight the three key aspects identified above in each of them and discuss whether their results can be used in our solution.

2.1.1 "Dynamic Steiner Tree Problem" [1]

The first paper we want to describe is the one named "Dynamic Steiner Tree Problem" [1]. It suggests ways to compute a Steiner tree with or without "re-arrangements", a rearrangement being the replacement of a route in the tree by another one. A route is a set of edges from one node to another. In the non-rearrangeable case, an added route can not be removed later during the computation of the tree, and will be present in the returned Steiner tree.

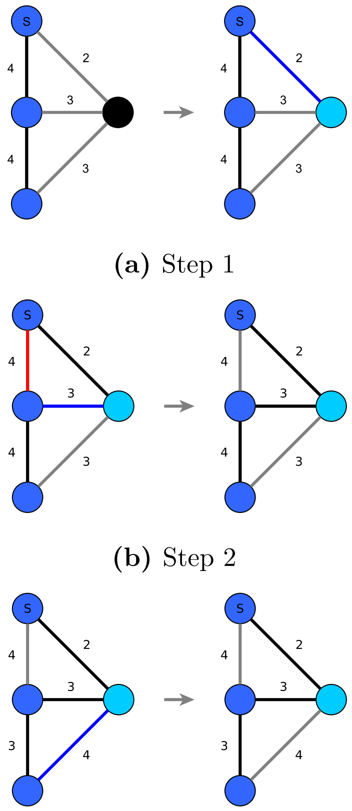
The dynamic aspect of the dynamic Steiner tree problem comes from the fact that the Steiner vertices to be added to the tree are not known in advance. In this context, the online characteristic of the problem can be modelled as having an ordered list of Steiner vertices and the need to add every vertex in the order given by this list.

Subscription management In both cases (rearrangeable or not), the algorithms present two functions: the addition function and the subtraction one. When adding a node to the tree, they use a greedy addition. Upon a removal, in the non-rearrangeable case, they remove the node from the tree only if it is of degree one; in the rearrangeable case, they also consider removing nodes having a degree of two, as well as the fact of swapping an edge for another in order to try to improve the tree.

Complexity The greedy approach is in $O(n)$ (n being the number of nodes in the tree), and is thus fast enough for what we want to achieve (cf. section 1.3.1, the removal meets this goal as well as it is done in $O(1)$.

Optimisation Regarding this matter, the case of interest for us of course the rearrangeable one, where it is possible to change a route in the network. To keep the tree competitive, rearrangements are performed during additions and subtractions. During an addition, the method is to create loops by adding one by one the paths from the newly added node to the other nodes of the tree and aim at, under certain conditions, removing the most expensive edge in the loop.

This operation has a complexity of $O(n \times m)$ (n being the number of nodes in the tree and m, the number of edges in the tree), and is illustrated on fig. 2.1. Following a removal, they try to optimise the tree when a node of degree two is removed by reconnecting the created components, this has a complexity of $O(n^2)$.



(a) Step 1

(b) Step 2

(c) Step 3

Nodes:

- A **blue** node is a Steiner node
- The **cyan** node is the node to be added at this iteration
- A **black** node is not a Steiner node

Edges:

- **black**: in the tree
- **blue**: added to the tree
- **red**: removed from the tree
- **gray**: not in the tree

Step 1 : At the first step, the black node is greedily added to the tree, as it becomes a Steiner node.

Step 2 : At this step, the blue edge is added as it connects a **blue** node with the **cyan** one. The **red** edge, being the most costly in the created loop, is removed.

Step 3 : This is the same process as in **Step 2**, but this time, the added **blue** edge is also more costly. It is thus not kept.

The process in this example is a bit simpler than in [1] as there is an additional constraint regarding the removal of the most expensive edge in the loop. Indeed, if the most expensive edge ($e1$) is different from the one that was added to create the loop ($e2$), the decision to remove $e1$ is taken only if $cost(e1) > \delta \times cost(e2)$, with $\delta \geq 1$.

Figure 2.1: Improvement through loop creation

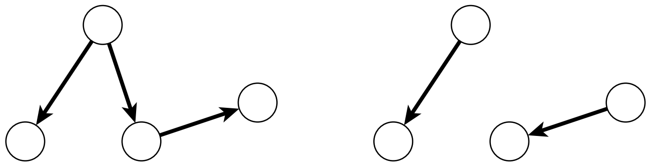
COMPONENT

In graph theory, a *component* is defined in an undirected graph. In such undirected graph, a component is a subgraph in which there exists a path from each vertex to each other vertex of this subgraph.

In our case, as we deal with directed trees, we call a component a tree that would satisfy the definition of a component if its edges were not directed. An illustration is given on fig. 2.2.

Limitations The algorithm provides a way for the tree to remain competitive with respect to the optimal Steiner tree for a given set of Steiner vertices.

A variable level of competitiveness is expressed via parameter δ of the algorithm.



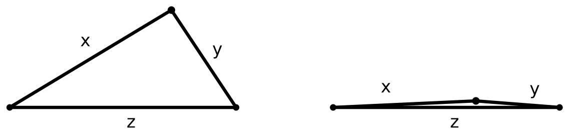
(a) One component **(b)** Two components

Figure 2.2: Illustration of components

Unfortunately, we cannot use their method as it is based on the hypothesis that the edges in the graph respect the *triangle inequality.*

TRIANGLE INEQUALITY

In a triangle, the *triangle inequality* is satisfied if the sum of the length of two sides is greater of equal to the length of the remaining side. An illustration is shown on fig. 2.3.

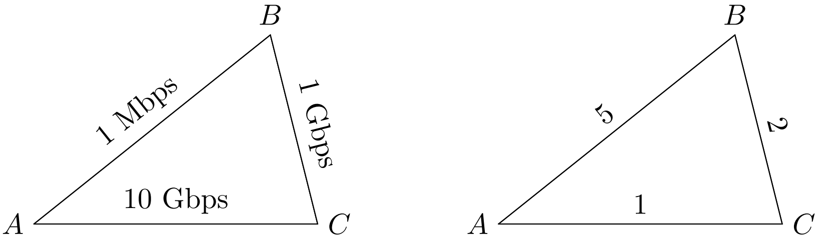


A triangle whose sides are x, y, z must satisfy the following rules to satisfy the triangle inequality:

- $x + y \geq z$
- $x + z \geq y$
- $y + z \geq x$

Figure 2.3: Illustration of the triangle inequality

In our networking environment, it is indeed possibly not the case that the edges of a network satisfy this inequality. A simple example is shows at fig. 2.4, going from A towards B directly would offer a small bandwidth of a 1 Mbps while going through C would offer 1 Gbps.



(a) Links with bandwidths **(b)** Subgraph

Figure 2.4: Links with weights

A second aspect that seems to limit the applicability of this solution in our case is that the amount of rearrangements required increases quadratically with the number of events occurring in the network, an event being either the addition or the removal of a Steiner vertex.

Finally, in the rearrangeable case, the additions are made in $O(n) + O(n \times m) =$

$O(n \times m)$ and the removal in $O(1) + O(n^2) = O(n^2)$. This scales badly on large topologies, such as the ones we are willing to consider.

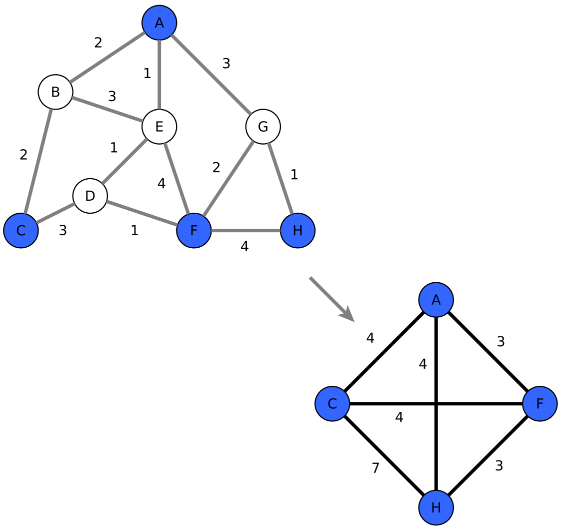
2.1.2 “Multicast Routing for Multimedia Communication” [2]

This paper proposes an algorithm to compute multicast trees for multimedia application, meaning that it aims at computing a Steiner tree while taking into account a *delay constraint* that is relevant on networks, the delay being the time for one packet to go from one node to another. It suggests to compute a *closure graph* on the subgraph composed of the subscribers and source nodes of the multicast group. This closure graph is a complete graph on those nodes where the cost of an edge (u, v) is the cost of the cheapest path respecting the delay constraint from u to v in the graph of the network. We must understand that under this rule, the cheapest path between two nodes may not be selected as it may cause a too large delay.

DELAY CONSTRAINT

CLOSURE GRAPH

An example of closure graph is depicted in fig. 2.5.



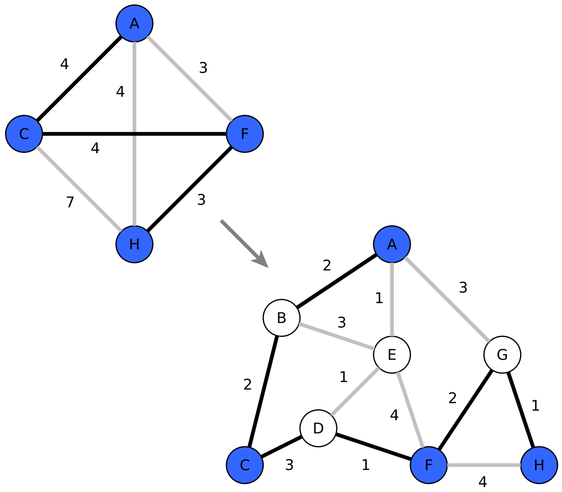
Nodes:

- blue nodes are the subscribers and the source
- white nodes are not be part of the closure graph

The closure graph (on the right) is a full-mesh graph linking blue nodes (from the left graph) together with their respective shortest path lengths. The delay constraint was omitted in this example, as it only constrains which shortest paths will be selected, and not the global process of building the closure graph.

Figure 2.5: Computation of a closure graph

Subscription management When building the tree, the algorithm proposed in [2] finds a spanning tree of the closure graph. Due to the fact that the closure graph only has the subscribers and the source as vertices, each addition to the tree requires the addition of one edge. When the computation of that spanning tree is over, it is expanded to return the computed multicast tree. Expanding the spanning tree means that for each edge (u, v), the corresponding path in the network graph is added to the solution. An example is shown on fig. 2.6



Closure graph (left):
Each edge represents a shortest path in the graph of the network. The black one are the chosen ones to build the spanning tree of the closure graph.
Network graph (right):
Nodes:

- **blue** nodes are the subscribers and the source
- white nodes are not part of the closure graph

Edges:

- **black**: in the tree
- gray: not in the tree

Figure 2.6: Expanding the spanning tree of the closure graph

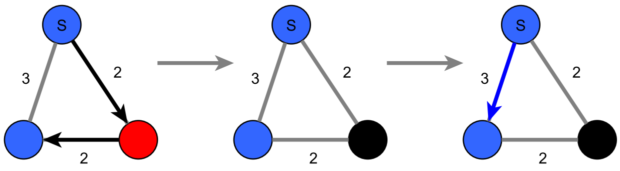
Complexity The closure graph has to be computed before starting the tree construction process. It therefore avoids to compute shortest paths during the execution.

Optimisation To obtain a competitive Steiner tree when the spanning tree on the closure graph is expanded, the algorithm uses several heuristics to order to select which node to add at each iteration. The process of building the tree is thus performed offline, because it needs to know the set of subscribers in advance.

Limitations Although the algorithm returns a multicast tree and is polynomial in complexity, it computes the tree in an offline way, because it needs to know in advance what the set of subscribers will be. This algorithm does not suit the online characteristic of our problem. The authors did not consider the leaves of subscribers (Steiner nodes removals) because they know the set of clients in advance and work in an offline environment.

2.1.3 "Fast Local Search for Steiner Trees in Graphs" [3]

A major issue in our problem is to find a way to remove nodes from the set of subscribers of the multicast stream. Removing a node from the set of subscribers makes this node a potential point of improvement. Indeed, there might exist another Steiner tree which does not contain this specific node. As shown on fig. 2.7, the branch of the Steiner tree that contained the removed node has be be replaced by another one, comparatively less costly.



Nodes:

- **blue** nodes are subscriber and source nodes
- the **red** node is the one that is removed
- a **black** node is not in the Steiner tree

Edges:

- **black**: in the tree, is removed with the **red** node
- **gray**: not in the tree
- **blue**: is added to reconnect the **blue** nodes

Figure 2.7: Change of the Steiner tree due to node removal

Subscription management [1] and [3] both describe a way to remove a Steiner vertex. The difference between them both is that in [1], upon the removal of a Steiner vertex, it is only removed from the tree if it is of degree 1 or 2. In [3], regardless of the degree of the vertex, it is removed, creating several components. Those components are then reconnected to get a valid Steiner tree.

The number of components to reconnect is equal to the degree of the removed vertex. Each of the created components is abstracted as a unique "*supervertex*" and a minimum spanning tree (MST) of the subgraph composed of those supervertices is computed. To compute that minimum spanning tree, they use the edges that connect the supervertices between themselves in the graph. If the computed MST is better than the solution which contains the removed vertex, that new MST is kept as the current solution. If it is not the case, the solution does not change and the removed Steiner vertex simply becomes a non-Steiner vertex while remaining in the tree.

To add Steiner vertices to the tree, they exploit the same idea as the one given in "Dynamic Steiner Tree Problem" [1] (see section 2.1.1), which is to add edges and break created cycles by removing the most costly edge in those cycles.

Complexity By using several data structures, their algorithm is able to compute a MST of the supervertices in $O(m \log n)$. The addition of vertices and the optimisation process are performed in $O(m \log n)$.

Optimisation The optimisation of the tree is done during the addition and removal of Steiner vertices as they reconsider the Steiner tree at each event.

Limitations In order to achieve the time complexity they advertise, they need to maintain lists of edges to speed up the computation of the MST of the supervertices.

They give no precise information on the space complexity of such data structures and on the time complexity to maintain them. Their approach would thus scale poorly when using large topologies.

2.1.4 "The Power of Recourse for Online MST and TSP" [4]

As stated in section 1.3.1, we want our solution to be efficient in an online network. To ensure this, we want to avoid rearranging the tree too often, as this process can have a significant complexity.

In [4], Megow, Skutella, Verschae and Wiese propose a way to avoid wasting swaps when computing an online minimum spanning tree (MST). A *swap*, in this context, is the replacement of an edge from the spanning tree by another one that was not in it, the resulting tree being a spanning tree of the same nodes. The problem of computing an online minimum spanning tree corresponds to finding a MST for a graph evolving node by node.

SWAP

Although we don't address the same problem, their goal is to build a tree in an environment very similar to ours: the construction cannot be done in polynomial time and must be done with a limited knowledge of what happens at the next step of the online process.

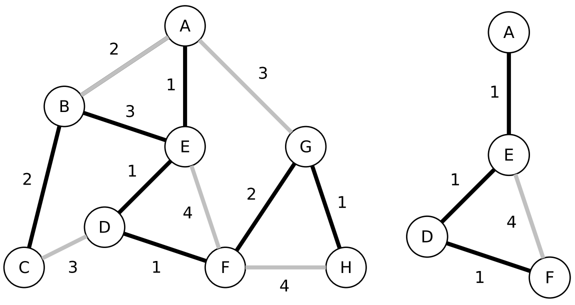
Optimisation Their algorithm basically adds new nodes greedily and then performs swaps to keep the tree as good as possible, meaning that they want to minimise its cost. In order to avoid performing too many of those swaps, they introduce the idea of *freezing rules*.
Two of these rules are described:

FREEZING RULES

- First rule: the goal of this rule is to avoid removing edges having a very small cost, as they are likely to be part of a good solution.
- Second rule: it is a more subtle one. It avoids removing an edge if that edge can be identified as part of the MST of a subgraph of the currently known graph so that that MST has a negligible cost compared to the current MST. This rule is depicted on fig. 2.8.

Limitations There are several reasons on why we cannot use those ideas as such:

- As said before, they address the problem of computing an online MST, which is a different problem from our case, where we need to build a Steiner tree. Not all the nodes from the network must then necessarily be in the solution tree.



(a) Graph with its spanning tree **(b)** Subgraph

Edges:

- **black**: spanning tree
- gray: not in the tree

The subgraph in fig. 2.8b can be extracted from the spanning tree in fig. 2.8a. If the cost of the MST if this subgraph is considered negligible with respect to the cost of the spanning tree of the whole graph, its edges can are frozen as they are likely to be part of the solution.

Figure 2.8: Second freezing rule

- Because they build a MST, they only need to consider (remove and add) edges one by one in order to improve the tree. Our case is different, as Steiner vertices might be connected via paths long of several edges. We therefore cannot, as it is the case with [4], operate one edge at a time.
- In their proof, they need the graphs to satisfy the *triangle inequality*, which is not necessarily the case in the set of graphs we consider. Such graphs rarely conform to this characteristic.
- In the online MST problem, the graph is discovered step by step, as the nodes to add to the MST are revealed. In our case, the whole graph is known and the only unknown variable is the set of nodes which indeed becomes Steiner vertices. We want to benefit from this by exploiting the knowledge of the graph we have.
- Due to the specifics of the problem they address, there is no discussion about the removal of nodes.

Their solution is thus not applicable to our problem, but we want to keep in mind the concept of *freezing rules* in order to avoid spending computation time on rearrangement that will probably not lead to a better solution.

2.1.5 "The Power of Deferral : Maintaining a Constant-Competitive Steiner Tree Online" [5]

This paper addresses the issue of improving the results of "Dynamic Steiner Tree Problem" [1] (see section 2.1.1). It provides a way to compute a Steiner tree that is $O(1)$*-competitive*, while performing only one *swap* per vertex addition. Here, a swap is, as presented while discussing [4] in section 2.1.4, the removal of a single edge and the addition of another. A $O(1)$-competitive tree can only be worse compared to the optimal tree by a constant factor.

$O(1)$-COMPETITIVE

This method suits what we want to achieve because it makes a limited number of changes in the multicast tree (typically, a single swap) after each addition of a Steiner vertex to the tree, corresponding to a join event. In the end, they provide a complete proof that such tree competitiveness can indeed be reached.

CLUSTERING
RANKING

Optimisation To be able to achieve their goal, they use two techniques: the *clustering* and *ranking* of vertices. Clustering is a way of grouping together vertices which are within a certain distance from one to another. Derived from those clusters are the ranks, which are then associated to vertices. They are based on the length of the tree edges. By setting these ranking values, the authors free themselves from using edges any more, and can focus on the vertices while building the tree.

VIRTUAL RANKS

In addition to those concepts, they add a third notion : the notion of *virtual ranks*. Those slightly differ from the *ranks* as they should provide an upper bound to the assigned *rank* of each vertex.

Based on this precomputed information, they reach their objective as they compute a Steiner tree that keeps its competitiveness while allowing only a limited number of swaps (one per addition). The small number of swaps needed is a very interesting characteristic for our online problem.

Limitations We reach the same conclusion as for papers presented in the previous sections. Even though their results are appealing, they can not be used for several reasons:

- As in [1] and [4], the proof is made under the assumption that the graph satisfies the *triangle inequality*, which is not necessarily the case in realistic networks.
- They do not consider the removal of Steiner vertices.
- At each addition of a node, some computation is needed on each vertex already present in the Steiner tree (clustering and ranking processes). However, we want to avoid too much computation when simply adding a node, so that the addition process takes a minimum amount of time.

As per these reasons, their results can not be applied to our problem. However, we need to keep in mind that minimising the amount of swaps/rearrangements is a goal we must pursue, as modifying the multicast distribution tree corresponds to editing the flow tables of each `OpenFlow` switch impacted by the change. Here, we want to avoid frequent changes being made to those flow tables.

2.1.6 Conclusion

In the end, none of the works we found provides a complete method that is adapted to the problem we want to address. The different works done have several flaws that can not be ignored in our case.

To summarise the limitations identified in the presented papers, the results of some papers can not be exploited because they require that the graph of the topology satisfies the *triangle inequality*. In some of the proposed solutions, there was *no mention on removing nodes* from the tree. While they achieve good competitiveness, their result is not guaranteed when considering the removal of nodes. When the mention was made, and both addition and removal where treated, improving the tree was done at the same time, resulting in operations needing *too much computation*. Finally, we need to develop a method that works in an online environment, and we therefore can not use results based on and exploiting an offline situation.

Nonetheless, the ideas we kept from the different papers are the following:

- The greedy addition of vertices to the solution.
- Swapping edges in order to get or maintain a better solution tree.
- Maintain a data structure on the side to avoid computation during the construction of the tree.
- Have a fast way of removing Steiner vertices.
- Avoid to consider rearrangements that will almost certainly not lead to an improvement of the solution.
- Try to minimise the number of rearrangements when trying to improve the tree to tend towards a better solution tree.

We used all those ideas in order to develop our method that is fully detailed in the upcoming chapter 3.

2.2 Integration into an `OpenFlow` network

In this section, we introduce papers discussing actual implementations of multicast into an `OpenFlow` network. We further comment their contribution relative to multicast in the `OpenFlow` world and discuss whether our approach can benefit from their breakthroughs.

The literature addressing the multicast problem in software-defined networks is still relatively short. The problem of computing competitive trees is usually not the main purpose of the papers. Instead, such papers focus on solving a restricted problem relative to multicasting in an `OpenFlow` network, while leaving other aspects aside : some focus on the implementation and demonstration of an actual set of features, while other attempt on implementing failure-recovery mechanisms. In each case, their multicast distribution tree computation process is either not scalable to larger networks (as the network graphs we used in our assessments), or not focused on finding close-to-optimal distribution trees (as they settle for regular path computation algorithms based on shortest paths).

A listing of such contributions related to the current tackled problem of multicast tree computation is developed, along with reasons why we think they do or do not solve this problem.

First, we describe more general discussions about what software-defined networks (SDN) allow in terms of flexibility and abstractions. A paper demonstrating ways of enriching the network interface with extended capabilities is described. Clean-slate approaches are by definition not compatible with the current stack of abstraction used in practice, either because they require specific non-standard protocols for their execution, and are therefore not integrable with current and widely used protocols such as IGMP [23], used for handling multicast groups, or because they define a new network stack on which applications must be ported.

Second, several so-called "clean-slate" implementations are described.

Third, a paper describing ways of handling failures in the network are described.

Finally, a conclusion is drawn regarding which aspects of the discussed contributions our presented approach could benefit from.

2.2.1 Clean-slate network abstractions/architectures

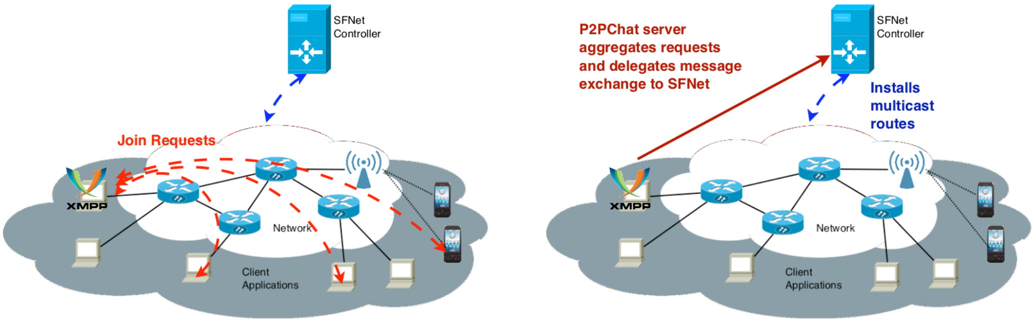
"Towards Software Friendly Networks"

"Towards Software Friendly Networks" [6] is a paper discussing how software-defined networks (SDN) could revolutionise the way network applications are built on the network, by enriching the current interface of the network, allowing for richer interactions. A network "operating system" can then run on top of the open and vendor-independent API which is OpenFlow, supporting different services such as for example multicasting.

The authors implemented a framework, called "SFNet", providing an API for software applications to interact with the network. Users therefore do not need to worry about network structure and low-level design for using and implementing services on top of these networks.

Their propose an implementation providing applications with an extended range of features, such as for example information on the congestion of the links. They also demonstrated a bandwidth reservation scheme which can improve how critical traffic needs to be prioritised in a network. The controller can be independently requested for such pieces of information, resulting in more intelligent applications and services.

Their last implementation example provides multicast capabilities to applications for a peer-to-peer (P2P) chat application where distribution trees are being set up between each user to every other (in a mesh-casting scheme). The message and video conferencing packets are being switched in the network directly, removing the need for them to be routed to a centralised XMPP server application. The network application stands there only to define routing instructions in the network, and is as such decoupled from the service itself, meaning that the XMPP server or the controller crashing does not affect currently set up chat sessions. Those interactions are depicted on fig. 2.9.



(a) Join requests to the XMPP server

(b) Requests forwarding to the SFNet controller and route installation on the switches

Figure 2.9: Configuration of the OpenFlow network (borrowed from [6])

The paper demonstrates how SDN can provide applications with more features and in a decoupled fashion.

2.2.2 Clean-slate implementations of multicast in an OpenFlow network

Two clean-slate implementations of multicast service are discussed in this chapter, the *Castflow*, and the *Multiflow* solutions.

“CastFlow: Clean-Slate Multicast Approach using In-Advance Path Processing in Programmable Networks”

“CastFlow: Clean-Slate Multicast Approach using In-Advance Path Processing in Programmable Networks” [24] presents a clean-slate approach for achieving multicast in an OpenFlow network.

In order to address the unpredictable nature of multicast trees, which is due to the potentially large number of join/leave requests occurring in the network, the authors chose a solution in which the computation of all possible routes from sources to group members is done in advance, such that those multicast events can be treated faster.

To support rapid changes in the multicast tree (rapid occurrence of join/leave events), the authors seem to have chosen to compute in advance all the possible path routes for a given multicast group. Their argument is that the full knowledge of the network topology and the pre-computation of optimal multicast trees could avoid situations where the trees could get destabilised, as it is the case with current distributed approaches, when facing a high rate of host joins and leaves.

Group management, on the other hand, is handled just as `IGMP` but doesn't use it in practice. One should plug a separate and specialised service of multicast group management into the algorithm.

The pre-computation phase calculates all possible routes between all possible multicast sources and hosts of the multicast group.
However, it is not reasonable to pre-compute close-to-optimal trees for every reachable situation. For example, consider a network composed of m sources and n potential clients. If we had to compute one close-to-optimal tree per possible $(source, clients)$ pair, we would end up with a number of trees to pre-compute of $m \times 2^n$. In our tests, we used topologies with one source and a number of potential clients ranging up to more than 700 nodes. Using a pre-computation approach would require us to find 2^{700} distribution trees for all situations, which makes any pre-computation step unrealistic.

To countermeasure this fact, the tree computation approach followed by Castflow does not yield close-to-optimal trees. Their method relies on the computation of a minimum spanning tree (MST) centralised on a multicast source node. From the topology graph of the network, they modify a copy of the graph where the weights of each edge is set as the distance from this edge to the multicast source node in the original topology graph. The MST is then computed on this separate graph.

Shortest paths from the source to each client node from the multicast group are then extracted from this MST. Redundancies from this list of links are then removed.

This method for building multicast trees does not solve the optimisation problem we address in this paper. Events are handled directly and a deterministic tree with respect to the events occurrence is computed, following the data that was pre-computed beforehand. This enables the Castflow controller to install the new paths in the network by making a difference list of the links that must be added or removed.

As such, the Castflow method does not solve the tree computation problem the way we see it, as their goal is not finding close-to-optimal multicast distribution trees. They are then able to pre-compute the distribution trees.

A prototype using OpenFlow was implemented. Topologies analysed were generated by the BRITE tool [25]. The Castflow controller is implemented in several modules, as well as a basic application running on the emulated network, using the multicast service.

Nowhere do the authors speak about the conformance of their prototype with any version of the OpenFlow specifications.

“Multiflow: Multicast clean-slate with anticipated route calculation on OpenFlow programmable networks”

“Multiflow: Multicast clean-slate with anticipated route calculation on OpenFlow programmable networks” [26] is an article introducing the Multiflow controller, which is, like Castflow, provides a clean-slate approach for building multicast distribution trees in an OpenFlow network.

For computing multicast distribution trees, the authors decided to use the Dijkstra algorithm [27]. As such, the method can not yield optimal distribution trees. Just as for Castflow, Multiflow does not focus on methods for finding close-to-optimal distribution trees, and rather uses a relatively low-complexity algorithm to achieve the tree building process.

Group management in Multiflow is handled differently than in Castflow. IGMP messages are actively matched and forwarded to the controller for it to handle multicast groups in a centralised fashion. The choice has been to follow the IGMP protocol such that the solution can be introduced in today’s networks, and directly compared with respect to performance against today’s networks which actively use IGMP for group management.

The goal of the paper is to measure the benefits of using OpenFlow to remove the propagation of IGMP messages through the network for building multicast distribution trees.

For that goal, Multiflow was compared to another controller, namely the “OpenMcast” controller, simulating a standard network in a SDN environment. IGMP queries are flooded between switches, just as it would be the case in a standard network when groups are being set up.

In this scheme, once groups are known in the network, clients spread IGMP join packets, which are then forwarded across the network up to the switch directly connected to the server, thereby creating a distribution path between the server and the client issuing the IGMP join requests. The behaviour followed by this controller is expected to be very much like what is being used today in networks that are not software-defined, therefore yielding a good comparison of the improvements of using software-defined networks (here with `OpenFlow`) to reduce the amount of group management packets in the network.

The paper does not make use of realistic topologies but rather uses tree-structured networks with a limited number of nodes.

The Multiflow controller greatly reduces the number of group management packets in the network as opposed to classical networks. Indeed, in the Multiflow approach, group queries are directly captured and forwarded to the controller and do not get propagated throughout the network.

Although the Multiflow solution is not aimed at solving our specific optimisation problem, we can still extract useful ideas from it, one of them being the conformance ideal to the `IGMP` protocol, which is achieved by forwarding `IGMP` packets to the controller. The controller, on the other hand, has to deal with such messages accordingly.

2.2.3 Failure-recovery schemes

"A design and implementation of OpenFlow Controller handling IP multicast with Fast Tree Switching"

"A design and implementation of OpenFlow Controller handling IP multicast with Fast Tree Switching" [28] describes an `OpenFlow` controller handling IP multicast along with a method for switching a multicast tree to another, enabling quick recovery in case of failure.

Redundant but group-uniquely identified trees are computed for each multicast group and then set up into an `OpenFlow` network.

The `OpenFlow` controller is made of several modules. Using one of those modules, the controller watches `IGMP` packets from devices and stores the clients locations in it.

Topology is inferred from using the LLDP protocol [29].

Two redundant trees are computed for each multicast group, one *active* and one *backup*. Their identification number is embedded into the packet header (for example, in the Ethernet destination address). The redundant trees can be link-protected with respect to the previously computed active tree.

The authors chose to cache multicast trees covering all switches of the network, so that multicast trees don't have to be computed after each receiver join or leave event.

The chosen method to compute multicast distribution trees was to use a Dijkstra algorithm [27]. This approach is unfortunately far from being optimal.

Before computing the distribution trees, the authors perform some link cost engineering. The first tree is computed with each link cost being (re-)set to a value of 1. The second redundant tree is computed after incrementing those previously used links by the cost of the whole previously computed tree. That is, if the cost of the active tree is $|E|$, then the edges used by that tree see their cost increased by $|E|$, thereby receiving a cost of $1 + |E|$. The second created tree is ensured to be different (except when there is no other alternative than going through the same edges). That way, the first *active* tree computed corresponds to a minimum spanning tree (MST).

We stress that their approach is meaningful only when link costs are very close to a single value, and does not optimise the trees in any way with respect to actual link cost. Indeed, their focus is not put on optimising the tree computation process, but rather to demonstrate how tree switching can be done in practice in an OpenFlow network.

In practice, the sender switch (the switch directly connected to the source of the multicast stream) has to rewrite the values of the packet header according to which tree it must be routed on (here, the Ethernet destination address is used as header field to match on). When a failure occurs, the controller only needs to edit the flow table of the sender switch in order to do a tree switching procedure. Receiver switches (switches directly connected to clients), on the other hand, should restore the correct header values (here, the Ethernet destination address of the source) to the original value for the packet not to denied at the client side.

The controller was implemented using the C language and is based on the Trema framework [30].

As our thesis does not focus on mechanisms for achieving failure-recovery, the contributions from this paper were of little use to us. However, some facts such as the feasibility of IGMP forwarding was noticed and later mentioned in section 6.3.1 discussing on how to conduct an actual integration in an OpenFlow network.

2.2.4 Conclusion

The separation between control and data plane of SDN enables more services to be provided to end-users. Such services include for example providing multicast.

We identified some useful aspects from the papers discussing actual integration in current network environments (with an OpenFlow network):

- The papers currently in the literature do not solve the optimisation part of the multicast problem. Rather, they focus on other problems such as failure-recovery, as in section 2.2.3, or discuss about benefits and assess about the actual feasibility of using software-defined networks approaches, as in section 2.2.2. They usually settle for suboptimal approaches when it comes to building multicast trees.

- It is conceivable and practically realistic to match and forward group management packets from existing group management protocols such as `IGMP` to the controller for centralised group events processing. It even has well sought implications on performance as it removes flooding from the network, as described in the Multiflow solution from section 2.2.2.

Chapter 3

Multicast tree computation algorithm

The tree building algorithm is first formalised in terms of different smaller subproblems. In a second time, each of those problems is solved independently.

3.1 Finding competitive tree

The problem of finding a competitive tree is the issue of finding a Steiner tree $T = (V_T, E_T)$ of a subset C of the nodes of a graph $G = (V_G, E_G, w)$. Those nodes are referred to as the "multicast" nodes. Node s is the root of the multicast stream. w is a matrix of weights such that $w_{i,j}$ is the weight of edge $(i, j) \in E_G$ with $i, j \in V_G$.
The following holds:

- $C \subseteq V_G$
- $s \in C$: source of multicast stream
- T is a multicast tree of (s, C) wrt. G, meaning:
  - T is a tree
  - $V_T \subseteq V_G$ and $E_T \subseteq E_G$
  - $s = root(T)$
  - $C \subseteq V_T$: the tree may contain nodes that are not in C
  - All leaves of T are in C

The tree should be kept at a near optimal cost (the smallest possible cost). The cost of tree T is defined as the sum of the weights of its edges :

$$cost(T) = \sum_{(i,j) \in E_T} w_{i,j}$$

As the cost of the tree should be kept as low as possible while making sure that all multicast nodes are part of it, we can deduce that all leaves of the tree should belong to C. In fact, if a leaf is not in the client set, it can be removed as well as the edge(s) connecting it to the tree.

3.2 General method

As our objective is to build a tree with a cost as low as possible and comply with the imperatives of a realistic network, we need both a quick way to handle changes such as subscriber addition and removal, as well as a way to ensure that the tree remains competitive even after such changes are applied to the tree.
Our method can then be split into three main components:

- The addition of nodes to the set of subscribers as quickly as possible.
- The removal of nodes from the set of subscribers as quickly as possible.
- An improving function to try and find an improved solution for the current set of subscribers.

The goal is to have independent components, such that an improvement of the tree can be made at any time during the execution. This provides more flexibility regarding the way the tree is built and where time is spent. The method will thus work differently in a context where there are lots of join/leave requests from subscribers per unit of time than in a context where such events occur less frequently.

This means that trying to improve the tree can be made more or less often and with less or more time, depending on how much time is available, or on the quality of the tree that is sought.

3.3 Finding a solution

In order to solve the global problem of finding a competitive tree in a graph described in section 3.1, we describe the problems of adding and removing subscribers (clients) into and from the tree, as well as the problem of improving the current tree as presented in section 3.2. We then further reduce it to a set of smaller problems.

3.3.1 `AddClient` as a sub-problem

To first tackle the problem of building the multicast tree, we want to consider a simpler problem that is adding one node of the graph to the tree.

Given graph G and a solution tree T for clients set C, we want to add node $n \in V_G \backslash C$ to the tree.

This is achieved by the function `AddClient(`G, C, T, n`)`. This operation on the tree should be efficient.
This function requires:

- G a graph
- C a set of clients
- T a multicast tree of (s, C) wrt. G
- n a node to add to the clients (and thus to the tree)

It outputs T', a multicast tree of (s, $C \cup \{n\}$) wrt. G.

3.3.2 `RemoveClient` as a sub-problem

The next considered problem is removing a client from the tree. Given graph G and a solution tree T for clients set C, we want to remove node $n \in C$ from the tree.
This is achieved by the function `RemoveClient(`G, C, T, n`)`.
This function requires:

- G a graph
- C a set of clients
- T a multicast tree of (s, C) wrt. G
- n a node to remove from the clients set (and thus from the tree)

It outputs T', a multicast tree of (s, $C\backslash n$) wrt. G.

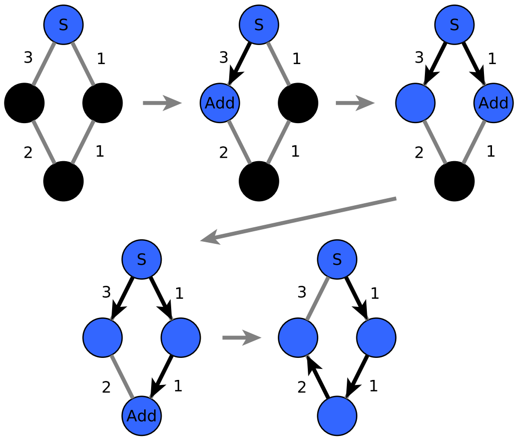
3.3.3 `ImproveTree` as a sub-problem

Adding a new node to the tree could create some room for improvement. In fact, the newly added edges could offer an alternative to some other edges being in the tree in order to find a better solution.

As shown on fig. 3.1, edges added to the tree in a specific order so as to connect all clients up to the last one (the bottom one). The last addition offers a rerouting possibility which, if taken, leads to a better solution.

In order to keep the cost of the multicast tree reasonable, we define a function `ImproveTree(`G, C, T, t`)` that aims at improving the tree (if such an improved tree exists).
The function requires:

- G a graph
- C a set of clients
- T a multicast tree of (s, C) wrt. G



Nodes:

- A **blue** node is in C
- A **black** node is in $V_T \backslash C$
- The node labelled “S” is the source
- A node labelled “Add” is a new addition

Edges (label is the weight):

- gray edges: not in E_T
- **black** arrows: in E_T

The nodes are added greedily at step 2, 3 and 4. From the 4th to the 5th step, the edge of weight 3 has been replaced by the edge of weight 2 to improve the tree. The cost of the tree is reduced by one unit of weight.

Figure 3.1: Improvement step leading to an improved tree

- t available time to improve T

After time t, it should return T', a multicast tree of (s, C) wrt. G such that $cost(T') \leq cost(T)$.

3.3.4 Solving the multicast tree building problem

By using `AddClient`, `RemoveClient` and `ImproveTree`, we can build the tree by performing the actions of adding and removing clients one by one to an initial tree T_0 containing source s only.

The improvement function can be called any time between any of those actions. This tree construction process is achieved by `BuildMCTree(`G, A, s`)`.
This function requires:

- G a graph
- A a list of actions (either an addition, a removal or an improvement)
- s the source of the multicast stream

The output is T, a multicast tree of (s, C) wrt. G, where set of clients C is defined as the clients that were added and not removed at the end of the list of actions.

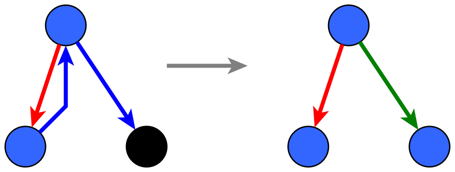
3.3.5 Solving `AddClient(`G, C, T, n`)`

The requirement expressed in section 3.3.1 defining the `AddClient` problem that a client should be added as quickly as possible, can be expressed as the need for a low time complexity function.

The proposed method is to greedily add node n to T.

Upon the addition of n to the tree, the node can either be already in the tree ($n \in V_T$), or n is not in V_T. In the first case, adding n to the set of client C is enough. In the latter case, we use a pre-computed complete graph $G' = (V_G, E'_G, w')$ where w'_{ij} is the cost of the shortest path between nodes i and j in G. To find the shortest path(s) between T and n, we need to look once at each node in V_T, and select the one that is the closest to n in G' in order to add it to the tree. The complexity of this operation is thus linear with the size of V_T.

As shown on fig. 3.2, we must pay attention to the fact that some edges in the graph might have a weight of zero. However, in real networks, such a case should never occur because having links with a weight of zero could cause the computation of the shortest paths to fail. In our case, if some edges had a weight of zero, the shortest path chosen and returned could create a loop in T. This can easily be avoided by *cleaning* this path so that only one node of T appears in it. This cleaning process is detailed in sections 3.3.13 and 3.3.14. As explained in the latter section, the complexity of cleaning such a path is linear with the size of V_T.



Nodes:

- A **blue** node is in C
- The **black** node is in $V_T \backslash C$, it is to add

Edges:

- **red**: in E_T, has a weight of zero
- **blue**: reconnection path that creates a loop
- **green**: path to add, same weight as the blue one

Figure 3.2: Loop creation due to zero-weight edges

In the end, the time complexity of this procedure is $O(|V_T| + |V_T|) = O(|V_T|)$, which is the sum of the two sequential parts of `AddClient`: the greedy addition and the cleaning of the path.

3.3.6 Solving `RemoveClient`(G, C, T, n)

The method we use is to simply remove node n from the clients set C and depending on the case, perform some changes on T.

In the specific case where client n to remove is of degree one, the branch on which the client was can be cleaned in the upstream direction.

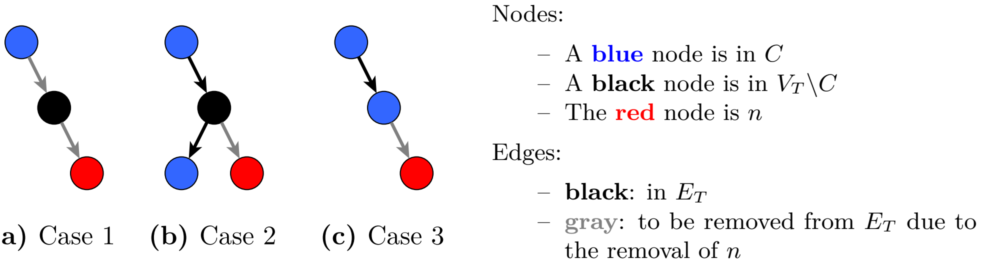
In fact, if the **degree of** n **is one**, it means that the node is a leaf, and that such a node can be removed without seeing the tree split into several components. A decision must be taken based on the state of the parent node of n, which falls into one of the three following situations:

Case 1: The parent is not in C and of degree one: it means that the parent is a leaf that is not in the set of clients of the multicast stream. It must thus be removed, and the decision of what to do next is raised on its parent. This is what we call in this situation a step of cleaning.

Case 2: The parent is not in C and has a degree strictly superior to one: the node is not a leaf and must thus remain in the tree. The cleaning is over.

Case 3: The parent is in C: the new tree matches the definition of a multicast tree as we defined it. The cleaning is over.

Those three cases are depicted in fig. 3.3.


Nodes:
– A blue node is in C
– A black node is in $V_T \backslash C$
– The red node is n
Edges:
– black: in E_T
– gray: to be removed from E_T due to the removal of n

(a) Case 1 **(b)** Case 2 **(c)** Case 3

Figure 3.3: Removal of a degree one Steiner vertex

Practically, this is achieved by the `AscendingClean` function, described in sections 3.3.10 and 3.3.11.

When the **degree of n is strictly superior to one**, there are two alternatives of action to carry out. The first one being to remove n from T and then trying to find an alternate path that returns a less costly tree, while the second one is to remove n from T and then leave T as it currently is. Improvement steps are thus left for later. We chose the latter solution because the removal of n would cause the need to reconnect the created (now disconnected) components (as many as the degree of n in tree T was) and would take time. We would rather keep `RemoveClient` fast and rely on `ImproveTree` to reconsider n later on and find a better path.

The complexity of `RemoveClient` is different in the two cases. When the degree of n is one, the algorithm performs in $O(|V_T|)$, which corresponds to the extreme case where there is only one client and the tree is made of a single branch from the source to this client. The whole branch must be then cleaned. In the other case, when the degree of n is strictly superior to one, the removal of n is done in $O(1)$ because no cleaning work has to be done on the tree.

3.3.7 Solving `ImproveTree`(G, C, T, t)

When trying to improve T, we use local search, making successive calls to `ImproveOnce` for a given amount of time t. The `ImproveOnce` function is described in upcoming section 3.3.8. In that scheme, successive calls of `ImproveOnce` are made on T.

We use the `simulated annealing` approach in order to avoid getting stuck in local optima. For that purpose, we introduced a *temperature*: the higher the temperature is, the more likely `ImproveOnce` will be allowed to deteriorate the solution if no improving tree can be found.

We have two possibilities to schedule temperature. Either we keep it constant throughout the `ImproveTree` process, which makes the probability to deteriorate also a constant, or we make the temperature decrease linearly with time. In the latter case, as time runs out, the search becomes more and more intensive and the `ImproveOnce` procedure is less and less likely to deteriorate the tree.

In order to avoid wasting calls to `ImproveOnce` that would probably not improve the tree, we initialise an empty `tabu list` before the first call to `ImproveOnce`. Its purpose and usage will be explained in section 3.3.9.

Once the allowed time is over, the best tree that was found during the time window is returned.

3.3.8 `ImproveOnce` as a sub-problem

This function aims at improving the tree after perturbing it. If an improvement can be found, the improvement is returned. If such an improvement cannot be found, a degradation is allowed with a probability derived from two parameters: the actual level of degradation and a given temperature.

Those three outcomes are integrated in the concept of *swap*, which we define as the removal of an edge from tree T and the addition of a new path between the two created components. This function performs at most one such swap and returns the possibly modified tree. SWAP

The function requires:

- G a graph
- C a set of clients
- T a multicast tree of (s, C) wrt. G
- *temp* the temperature from which to derive the probability to deteriorate the tree
- *tabu* a list of edges that cannot be removed from the tree

It outputs T', a multicast tree of (s, C) wrt. G. The cost of T' can be lower, higher or the same than the cost of T.

3.3.9 Solving `ImproveOnce`$(G, C, T, temp, tabu)$

This function should perform a swap as defined in the preceding section. We describe `ImproveOnce` in two steps: first we talk about the choice of an edge to remove, and the heuristics we use to do this choice. Then, we describe how we reconnect the components created by the removal of the chosen edge. The output of this function is a tree T'.

Algorithm 1 `ImproveOnce`$(G, C, T, temp, tabu)$

```
repeat
  $e \leftarrow$ select edge from $T$
until $e \notin tabu$
$p \leftarrow$ path containing $e$ in $T$
remove all edges in $p$ from $T$
$p_r \leftarrow$ reconnection path
if $|p_r| < |p|$ then
  $p_n \leftarrow p_r$
else
  {reconnection path $p_r$ is degrading}
  $\mathcal{P} \leftarrow$ probability to degrade $(p, p_r, temp)$
  $degrade \leftarrow$ allow degradation with probability $\mathcal{P}$
  if $degrade$ then
    $p_n \leftarrow p_r$
  else
    $p_n \leftarrow p$
  end if
end if
reconnect $T$ with $p_n$
```

Choosing an edge to remove

Which edge to select for removal can be done in several ways. This choice is determined by a *heuristic function.*

HEURISTIC FUNCTION

In any case, an edge cannot be selected if it appears in the `tabu list`. This list contains the edges that were recently added to the tree by a reconnection. The initial `tabu time-to-live` (TTL) of an edge in the `tabu list` depends on the initial configuration of the algorithm.

At each call to `ImproveOnce` done during `ImproveTree`, the value associated to each edge in the `tabu list` is decreased by one. When it reaches zero, the edge is removed from the `tabu list` and becomes eligible again.

The three heuristics we defined and used for selecting edges to remove are the following:

- `RANDOM`: An edge is selected at random among the edges of tree T.

- `MOST EXPENSIVE EDGE`: The most expensive edge in the tree is selected (considering it is not in the `tabu list`). This choice is done in $O(m)$, m being the number of edges in the tree.
- `MOST EXPENSIVE PATH`: The choice of an edge made by this heuristics relies on the notion of path in the multicast tree.
  Here, we define a *path* as a set of adjacent edges $\{(n_0, n_1), (n_1, n_2), ..., (n_{n-1}, n_n)\}$ PATH so that:
  - $\{n_1, ..., n_{n-1}\}$ are of degree 2 in T
  - $\{n_1, ..., n_{n-1}\}$ are not in C
  - n_0 and n_n are either in C or of degree > 2

  The edge is chosen as being part of the p most expensive *disposable paths* in the tree. A path is disposable if none of its edges appear in the `tabu list`. DISPOSABLE PATHS
  p is specified in the configuration and serves as a source of diversity in the algorithm.
  The complexity of choosing an edge among the most expensive paths is done in $O(m)$, as paths defined as such can be computed by visiting each edge only once, at the expense of an increase space complexity. This selection among the most expensive paths from the tree will be described in section 4.2.3, discussing about our implementation choices. By keeping track of all the paths in the tree, the most expensive path can then be found in $O(1)$.
  However, we must take the `tabu list` into account. In the extreme case where all edges from the tree are in the `tabu list` and all paths are made of only one edge, no disposable path can be found. This limitation raises the complexity to $O(m)$ in this worst case scenario.

Replacing the removed edge

Upon the removal of an edge, two components T_1 and T_2 are created, T_1 being the *upstream* component containing the root of T. T_2 is consequently named the *downstream* component. Those must be cleaned as some edges might have to be removed. This process of tree cleaning is detailed in section 3.3.10. UPSTREAM DOWNSTREAM
In order to find a path to reconnect the two components, we visit the pairs u, v with $u \in V'_{T_1} \subset V_{T_1}$ and $v \in V_{T_2}$ in order to find a path whose cost is lower than the cost of the path that was selected. To search for that reconnection path, several parameters can be set:

- The **size of the search space**: sets the size of V'_{T_1}, the nodes in V'_{T_1} are a random sample of V_{T_1}. Of course, the size of V'_{T_1} is bounded by the size of V_{T_1}.

- The **search strategy**:
  - `BEST IMPROVEMENT`: all the pairs (u, v) with $u \in V_{T_1'}$ and $v \in V_{T_2}$ are visited.
  - `FIRST IMPROVEMENT`: the iteration is made on nodes $u \in V_{T_1'}$. For each node u, all nodes $v \in V_{T_2}$ are visited. The exploration of V'_{T_1} stops as soon as an improving reconnection path (compared to the previously removed one) is found. The remaining nodes of V'_{T_1} are not visited. Still, all nodes from V_{T_2} are visited for the current node $u \in V'_{T_1}$.

If an improving path is found from u to v, it is installed in the tree.

Regarding the time complexity of the reconnection procedure: the shortest paths are precomputed. They must not be done at this step. Let $S = |V'_{T_1}|$ (the size of V'_{T_1}) be the size of the search space. The complexity of looking for a reconnection path is then $O(S \times n)$.

We provide two ways of handling the reconnection if no improving path can be found, that is, if only degrading paths were found: either allow or disallow the degradation. This choice is made using boolean parameter `intensify-only`, which can systematically disallow degradations:

- `True`: the path that was removed is restored. Degradations are not allowed.
- `False`: the path that degrades the tree the least is considered. The probability to add the degrading path is then evaluated based on the temperature value *temp* and on the degradation. The degradation is based on the weight difference between the path that was removed and the degrading path replacing it. The probability p to degrade the tree can be expressed as such:

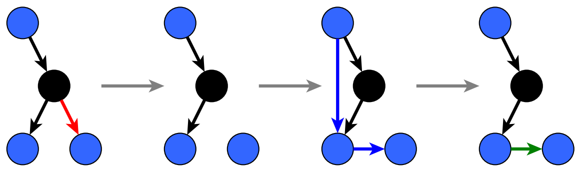
$$p = e^{\frac{-\Delta w}{temp}} \quad with\ \Delta w = \frac{weight(ReplacementPath) - weight(RemovedPath)}{weight(ReplacementPath)}$$

Setting this parameter to `False` enables the simulated annealing meta-heuristic introduced in section 1.3.3.

Whether the added path improves the tree or not, its edges are put in the `tabu list` and they become ineligible for a given time. This time is defined by the initial `tabu time-to-live` value, as defined in section 3.3.9.

When adding a path to the tree, that path might need to be cleaned in order to avoid loops. As shown on fig. 3.4, the reconnection path might go through nodes that are already in one of the components. To avoid creating loops, the reconnection path (**blue** edges) must be reduced (to the (**green** edge) such that it connects one and only one node per component. This is performed by the

`PathCleaning` procedure described in section 3.3.13. Doing so can only reduce the cost of the reconnection path.



Nodes:

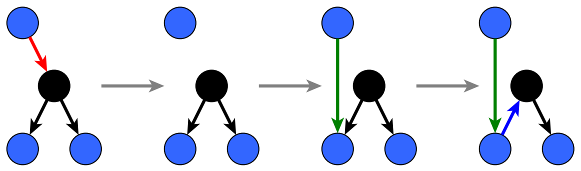
- A **blue** node is in C
- A **black** node is in $V_T \backslash C$

Edges:

- **black**: in E_T
- **red**: in E_T, to be removed
- **blue**: reconnection path
- **green**: path to add

Figure 3.4: Reconnection path cleaning

Another problem that occurs when reconnecting is that the root of the downstream component T_2 might to change. Figure 3.5 shows an example of this problem. Upon the removal of the **red** edge, the **black** node is left as the root of component T_2. If the **green** reconnection path is added, for the result graph to remain directed from the root to the leaves (and thus to remain a valid tree), a **blue** node must become the new root of the downstream component T_2. We therefore need to *reroot* the other component when improving the tree, this *rerooting* step is detailed in a further section 3.3.15.



Nodes:

- A **blue** node is in C
- A **black** node is in $V_T \backslash C$

Edges:

- **black**: in E_T
- **red**: in E_T, to be removed
- **green**: reconnection path
- **blue**: edge changed due to reroot

Figure 3.5: Illustration of the rerooting procedure

3.3.10 Tree cleaning as a sub-problem

After the removal of an edge, as only nodes in C are required to be in T, we want to make sure that no unnecessary node remains in components T_1 and T_2 before reconnecting them. This is performed by the `AscendingClean` and `DescendingClean` procedures.

As shown on fig. 3.6, upon the removal of an edge in the tree, a black node might become a leaf of the top component T_1 and a black node might become the new root of the downstream component T_2. Those nodes should be removed such that both the leaves of T_1 and the root of T_2 are client nodes.

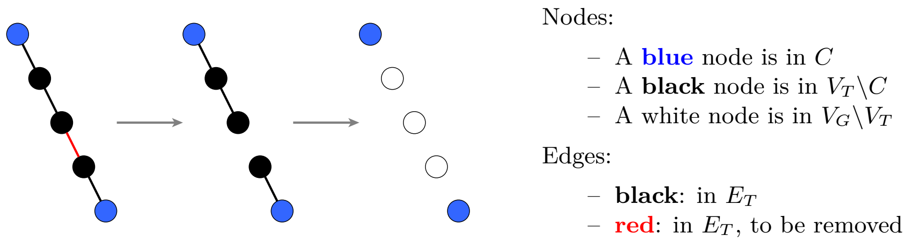

Nodes:
– A blue node is in C
– A black node is in $V_T \backslash C$
– A white node is in $V_G \backslash V_T$
Edges:
– black: in E_T
– red: in E_T, to be removed

Figure 3.6: Illustration of a tree cleaning procedure

The function `AscendingClean(`T_1, n, C`)` aims at cleaning component T_1, rooted at the source s.
It requires:

- T_1 a tree rooted at s
- All leaves $\neq n$ belongs to C
- n is a leaf, not necessarily in C

`AscendingClean` returns a tree T_1' such that:

- T_1 covers T_1'
- All nodes belonging to $T_1 \cap C$ are in T_1'
- All leaves of T_1' belongs to C

`DescendingClean(`T_2, n, C`)` aims at cleaning the downstream component T_2, which does not contain root s.
It requires:

- T_2 a tree rooted at n
- All leave belongs to C
- n is possibly not in C

`DescendingClean` returns a tree T_2' such that:

- T_2 covers T_2'
- All nodes belonging to $T_2 \cap C$ are in T_2'
- The root of T_2' either belongs to C or has a degree strictly superior to 1.

It should return a tree T_2' such that its local root is either in C or has a degree of at least 2.

3.3.11 Solving `AscendingClean(`T_1, n, C`)`

We can solve `AscendingClean` in a recursive way: to clean T_1, the function should check if $n \in C$ or if $degree(n) \geq 2$. If it is indeed the case, it should return T_1, or `AscendingClean(`$T_1 \backslash n, parent(n), C$`)` otherwise.

If the tree contains only one client, the case might arise that all edges from the tree must be cleaned. The worst-case complexity is thus in $O(m)$ with m being the number of edges of the tree.

3.3.12 Solving `DescendingClean(`T_2, n, C`)`

To clean T_2, if $n \in C$ or $degree(n) \geq 2$, return T_2. Otherwise, it should return `DescendingClean(`$T_2 \backslash n, child(n), C$`)`.

Similarly to `AscendingClean`, the complexity of this function is $O(m)$ for the same reasons.

3.3.13 Cleaning paths as a sub-problem

We sometimes need to clean the path to be added to the tree in order to avoid creating loops. Loops can be created in two cases, either due to edges having a weight of zero, or to the chosen reconnection path.
To achieve this goal, the `PathCleaning` function requires:

- V_{T_1} a set containing the vertices of the component T_1 that contains the root of T.
- V_{T_2} a set containing the vertices of the other component T_2.
- p a path (ordered list of vertices) from T_1 to T_2 such that its first vertex is in V_{T_1} and the last one is in V_{T_2}.

The returned value is p', a path from T_1 to T_2 such that only the first node of p' is in V_{T_1} and only the last one is in V_{T_2}.

3.3.14 Solving `PathCleaning(`V_{T_1}, V_{T_2}, p`)`

In order to keep only one node of each set of vertices V_{T_1} and V_{T_2} in the path, we go through the path from its end to its beginning until encountering $v_1 \in V_{T_1}$. Then, we go through p from the position of v_1 to the end until encountering $v_2 \in V_{T_2}$. p' is the sub-path from p from v_1 to v_2.

In the worst case scenario, which is the one where $p' = p$, we need to go through the path twice, the time complexity is thus $O(|p|)$ where n is the length of p.

3.3.15 Rerooting as a sub-problem

As shown before in section 3.3.9, fig. 3.5, a component, which is in itself a tree, might need to be rerooted such that the multicast tree, after reconnecting, stays consistent.
The function requires:

- T a tree
- r the new root of the tree

It returns T', a tree with the same nodes as T such that for each edge (u, v) in T, either (u, v) or (v, u) is in T'.

3.3.16 Solving `Reroot(`T`, `r`)`

To compute the rerooted tree T', the direction of each edge on the path from r to the current root of T must be inverted. In the worst case, the reroot function would have to go through all the edges of T, making the complexity $O(m)$, m being the number of edges in T.

3.4 Algorithm parameters

The algorithms presented in the above sections must be tuned with several parameters. Those parameters are summarised in this section, and their role in the local search approach is explained. References will be made to preceding sections where they have been first introduced.

`maximum improve time` : the maximum allowed time for a run of the `ImproveTree` procedure constrains the local search approach. It defines how much time this procedure will have to try and improve the current solution. This parameter is introduced in section 3.3.3.

`improve period` : this parameter was not introduced in the description of our method. When managing multicast on a network, we want to be able to call `ImproveTree` when needed. For that matter, it is possible to set a period that may be adapted to either improve on an event (join/leave) count basis or on a time basis.

`edge selection heuristic` : the edge selection heuristic is a very important piece of the algorithm. It defines which edge to select and reconsider in each run of an `ImproveOnce` iteration. The several edge selection heuristics that we implemented are further described in section 3.3.9. They include a `RANDOM` approach for selecting an edge from the tree, a `MOST EXPENSIVE EDGE` approach for selecting the most expensive edges first, and a `MOST EXPENSIVE PATH` approach for selecting paths with the most expensive cost first.

`tabu time-to-live` : in order to avoid considering the same edges for removal over and over, edges that are added to the tree are also stored in a `tabu list`. The *time-to-live* value determines how much time an edge is protected from the selection heuristic, that is, how many calls to `ImproveOnce` we should wait for such an edge to become eligible again for removal. This procedure, as well as its concepts were described in section 3.3.9.

`temperature schedule` : the temperature scheduling, which defines how temperature evolves over time, directly impacts the chosen simulated annealing approach. The scheduling of the temperature can either be `constant`, or `decrease linearly` over time. This parameter is used for solving the `ImproveTree` problem, as described in section 3.3.7.

p **`maximum selected paths)`** : the maximum number of *disposable paths* to consider when selecting edges to reconsider during the `ImproveOnce` procedure, this parameter is only useful when using the `MOST EXPENSIVE PATH` edge selection heuristic. *p* is introduced in section 3.3.9. It serves as a source of diversity in the algorithm.

`improve search space` : the improve search space argument defines the size of the random sample of nodes from which to try to find a reconnection path between the two created components, in an iteration of the `ImproveOnce` procedure. The search space size can either be a small constant (e.g. 5), or take the whole search space size. This parameter is further described in section 3.3.9.

`search strategy` : the search strategy to use when trying to find a better reconnection in an iteration of the `ImproveOnce` procedure can be of two kinds: either it returns the first path that is better (lower) than the former path to replace as soon as it is found (`FIRST IMPROVEMENT`), or it proceeds to search for the whole search space for finding the best one (`BEST IMPROVEMENT`). The used search strategy is described in section 3.3.9.

`intensify-only` : when trying to improve the tree, we might want to go for a solution that never degrades. For that purpose, that argument can be set to boolean value `True`. In that case, as described in section 3.3.9, when no improve path is found, the one that was selected and then removed is restored. The solution remains unchanged by this call to `ImproveOnce`.

3.4.1 Sources of diversification

As we are using local search to try to improve the Steiner tree, we want to bring diversification to the algorithm to enable it to examine new regions of the search space, without getting stuck into local maxima.

In our approach, several parameters bring diversification:

p **`maximum selected paths`** : if this value is set to one, the algorithm always selects the most expensive path in the tree for removal. At this point, we have a deterministic choice. We bring non-determinism by allowing to randomly select among the *p* most expensive paths and thus allow the search to explore other parts of the search space.

`improve search space` : when trying to reconnect the components created by the removal of an edge, we can bring diversity by considering a restricted search space for the problem of finding a reconnection path. To reduce the search space, we consider for reconnection a random sample of the nodes from the upstream component.

`search strategy` : for the same reason that a restricted `improve search space` brings diversity, using the `FIRST IMPROVEMENT` strategy brings more diversity than `BEST IMPROVEMENT`.

In fact, using `FIRST IMPROVEMENT`, we could miss the best improvement. That brings a non-deterministic aspect that leads the search to other regions of the search space than if we always went for the best improvement

`temperature schedule` : the temperature in the simulated annealing is a major source of diversification. When no improvement is found, the solution is more likely to be degraded when the temperature is high. When having the temperature decrease linearly with time, this corresponds to the beginning of the improvement step. It is then possible to escape from potential local optima and explore other parts of the search space that would have been unreachable otherwise. The `temperature schedule` therefore plays an important role in bringing diversification.

Chapter 4

Algorithm implementation

4.1 Technologies

In this section, we describe the technologies we decided to use and base our implementation [31] on.

4.1.1 Language and frameworks

Python 2.7

We chose to use Python version 2 [32] because of the rich ecosystem of libraries coming along with the language. Some libraries have been used. They are described in the upcoming sections.

We are familiar with the language as we used it before on several projects related to courses given by our supervisors.

Developing on Python is relatively straightforward. We believe that Python is a quick and appealing language for building an algorithmic solution from scratch. It is also easily portable on different systems.

In order to import and export binary files to and from memory, which is used for loading shortest paths in memory, we used the `pickle` Python module [32].

We tested our solution on both Mac OS X and GNU/Linux.

`NetworkX` framework

The data structures we based our implementation on include data structures from the `NetworkX` framework [33]. `NetworkX` provides a set of data structures including, amongst others, a `Graph` and a `DiGraph` structures, which provide everything we needed in terms of basic functionalities. As it is detailed in section 4.2, we extended those data structures to our needs by adding variables and methods. Also, some additional methods not directly related to the finding of a solution were defined, such as methods and formats for easily importing and exporting graph structures, which enabled us to build a highly configurable and easy-to-interface-with solution.

4.2 Main data structures

In this section, we present the data structures we used to implement our method. We explain their purpose and describe the information they hold, as well as the algorithm they implement.

Our two main objects are the `network graph` and `multicast tree` objects. A `multicast tree` is a dynamic structure representing the state of a multicast distribution tree. Each `multicast tree` is based on a static `network graph` data structure, representing the state of the network. `multicast tree` potentially makes use of several other data structures such as the `PathQueue` for ordering the paths currently in the tree.

Each of the aforementioned data structures is explained in the following sections.

4.2.1 Network graph

NETWORK GRAPH

The ***network graph*** data structure is a central piece of our algorithm. It consists of a static structure containing all information about the network and its graph representation. This structure holds the shortest paths in memory and is responsible for handling informations about the graph. Its actual implementation is an extension of `NetworkX`'s `Graph` object, thereby inheriting all of its methods.

Shortest paths (not to be confused with our concept of *path*) are stored in the `network graph` using several data structures:

`shortest paths` : a two-dimension array storing lists of edges between each two nodes in the graph;

`shortest paths length` : a two-dimension array storing the cost of such shortest path between each two nodes in the graph.

The value to use as the weight of the edges in the calculations can be configured in several ways, directly from an attribute or deduced by a method:

`WEIGHT` : the attribute `'weight'` is already defined as edge attribute in the topology file, this value is then used.

`GEO` : if geographic coordinates are given for each node, a value for the weight to give to each link can be derived from those coordinates. Typically, node physical distance is evaluated and used as weight attribute. This conversion is handled by the `Haversine distance` algorithm, described in section 4.3.1.

`NONE` : unit costs are set and used for each edge in the graph.

One must make sure that the pre-computed shortest paths loaded in memory used along with the topology are in sync with the attribute that was used for computing them. We provide a specific file format for keeping those files in sync.

4.2.2 Multicast tree

MULTICAST TREE

The ***multicast tree*** structure is based on the directed graph structure `DiGraph` of `NetworkX`. It is responsible for all the modifications and operations doable on the tree. It holds additional data structures for representing a multicast tree and performing improvement steps.

CLIENTS SET

One of the main instance variables is the ***clients set***. It is a set of nodes which represents the nodes currently part of the multicast group, that is, the subscribers of the multicast tree. When thinking of the multicast tree as a Steiner tree, nodes in the `clients set` correspond to the Steiner nodes.

The available methods for changing trees are the following:

`addClient` : the addition of a node to the tree is made greedily (cf. section 3.3.5). If the client to add is not already part of the tree (if the distribution tree does not cover it already), we loop on the nodes in the tree and select the closest one to the new client node. This shortest path is added to the tree by adding all the edges forming it. As it has been stated in the corresponding description of the function, the path needs to be cleaned in order to prevent loops that may arise due to edges having a weight of zero. However, zero-weight edges should not be used in practice. We still considered this case for robustness. When this cleaning is done, the new node is added to the `clients set`.

`removeClient` : as stated in section 3.3.6, we try to avoid disturbing the tree when a client is removed. The tree is changed only when the node to remove from the tree has a degree of one. We do that by using the `ascendingClean` method (cf. section 3.3.11), starting from the removed node. The node is then removed from the `clients set`.

`removeEdge` : the method first selects an edge and then possibly removes it in order to try and improve the tree. As explained in section 3.3.9, we implemented several ways of selecting an edge to remove:

`RANDOM` : a random edge is picked among the edges of the tree.

`MOST EXPENSIVE EDGE` : we loop on every edge of the tree and keep the most expensive one.

`MOST EXPENSIVE PATH` : we choose among the most expensive paths from the tree. For this, we maintain an ordered list of paths currently in the tree. This list of paths is ordered according to path weights and named `PathQueue`. This additional data structure is detailed in section 4.2.3. Parameter p (`maximum selected paths`), described in section 3.4, determines the number of most expensive paths to select from the `PathQueue`. One of those p paths is then randomly selected for removal. To remove a path from the tree, we remove an edge from the path and then apply the cleaning method, as we do in the two previous cases of edge selection.

Every time an edge of a path is chosen (as in the two latter cases), we check whether it can be removed by first checking the `tabu list` (whose purpose is explained in section 3.3.9).

`treeCleaning` : the cleaning of the tree has to be performed when an edge has been removed (cf. section 3.3.10). It consists in two function calls: `ascendingClean` on the created component that contains the source of the multicast stream and `descendinClean` on the other created component. Those functions work recursively.

`reconnection` : the method is called for reconnecting the disconnected components that were created by the `removeEdge` method (cf. section 3.3.9). To select a path to reconnect the components, we loop on a subset of nodes from the upstream component, containing the root of the multicast stream. The size of this subset can be defined using parameter `improve search space` (described in section 3.4). This subset of nodes is selected at random. For each of those nodes, we loop on all the nodes of the other component.

Depending on binary parameter `search strategy`, we break the first loop on the randomly selected nodes as soon as finding a better reconnection path, or we don't. Throughout the iterations of this loop, the best path found (strictly better than the sum of the weights of the edges that were selected and removed) is kept, as well as the less degrading one (this one can not be the path that was removed). At the end of the loop, if an improving path has been found, it is cleaned, as depicted in section 3.3.14, and added to the tree.

If no improving path is found, the least degrading one is considered. The probability to install that degrading path is evaluated based on the temperature value of the improvement step we are currently in. The higher the temperature, the more likely the degrading path will be added to the tree. The evaluation of this probability based on the temperature was explained in section 3.3.9. If the degradation is accepted, the path is cleaned and added to reconnect the two components.

Following the probability, if the degrading path is not kept, then, the path composed of the edges that were removed is re-installed.

`reroot` : method for updating the edges of a component when the root of that component is changed (cf. section 3.3.16). The rerooting is done on a component, which is a tree in itself. To do so, we start from the node that must become the new root and go in the upstream direction until reaching the old root. This is done by going through the `predecessors` of the new root in the tree, until the old one is reached. Because the component is a tree, there can be at most one predecessor for each node that is visited. We reverse each encountered edge by removing it and adding its inverse to the tree structure.

As mentioned above while explaining the selection of an edge to remove, the `tabu list` is also an instance variable of the `multicast tree`. Every time a reconnection is done, the reconnection path that is added to the tree is also added to the `tabu list` (even if it is the same as the removed one). Adding a path to the `tabu list` means adding each edge composing it. The `tabu list` is a dictionary where edges are keys and `tabu time-to-live` (ttl) are values (cf. section 3.3.9). After each call to `ImproveOnce`, the values associated to each key is decremented by one. If a value reaches zero, the associated key (representing an edge) is removed from the `tabu list`.

4.2.3 PathQueue

PathQueue

The ***`PathQueue`*** is a python priority queue that stores paths currently in the tree, ordered according to their weights.

It is used when choosing a path to be removed in order to try and improve the tree. It is only used with the `MOST EXPENSIVE PATH` selection heuristic. Paths are sorted so that the first path returned by a `pop` operation on the queue is the most expensive one. The motivation to create this data structure was to speed up the choice of a path during an improve. Another way of identifying the paths would be to go through the whole tree each time a path must be chosen. That operation is performed in $O(m)$, m being the number of edges in the tree. Paths would have to be re-computed each time. The `PathQueue` avoids doing so.

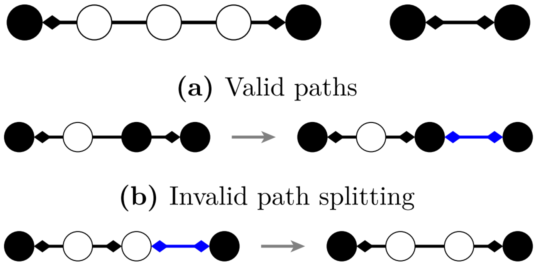
Paths are added to the queue in two occasions. First, when a client is added to the tree, the branch that is used to connect it to the tree is added to the queue. The second case is when a reconnection is made during an improvement step. The reconnection path that reconnects the two then disconnected components is also added to the queue.

VALID PATHS

Ideally, the paths in the queue should always be *valid paths*: every path in the queue should satisfy the definition of a *path* given in section 3.3.9. That means that the nodes at the end of the path either are clients and/or have a degree different from two. The other nodes of the path must all have a degree two and be non-clients. We call an invalid path a path that does not satisfy this last condition.

In practice, path validity is computationally expensive to maintain in the queue. Instead, we decided to fix invalid paths in the `PathQueue` only when such paths are "popped" from the queue. Two modifications can then be applied to fix invalid paths from the `PathQueue`: the `split` and `merge` operations.

Illustrations of *valid paths* and of `split` and `merge` are given on fig. 4.1. The explanations of the concept of `split` and `merge` come here-below.



(a) Valid paths

(b) Invalid path splitting

(c) Invalid paths merging

Nodes:

- A **black** node is either a client or has a degree $\neq 2$
- A blank node is not a client and has a degree of 2.

Edges: the diamonds represent the ends of a path in the `PathQueue`. Each path has a different colour.

Figure 4.1: Illustration of valid and invalid paths

SPLIT

A *`split`* should be performed when a path "popped" from the `PathQueue` is found to be invalid , which amounts to the situation on fig. 4.1b.

This path should be split into two paths around the node that breaks the validity property. Paths that need to be split are placed higher in the queue, as their real weight overestimates the smaller valid paths composing it. As such, it is not always mandatory to `split` such invalid paths.

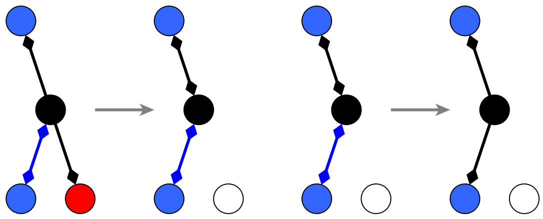
MERGE

Aside from the `split` operations, we also need to be able to *`merge`* two paths together when necessary. Unlike `split` operations, `merge` operations must be made as soon as possible, as the weights of the two paths that need to be merged underestimate the weight of the merged path. Such invalid and "unmerged" paths would then be placed lower in the queue, which would then break the ordering constraint of the priority queue. Indeed, if two paths become one in the `PathQueue`, the merged path is more likely to be chosen for removal and improvement.

We handle the `PathQueue` in such a way that no two paths from it cover each other.
The situations where splits and/or merges are carried are the following:

When removing a client : if the removed client is of degree one or two:

- *Degree 1*: the removed client is a leaf of the tree, and thus, a branch is removed. The paths in the `PathQueue` might need to be split and merged, as shown on fig. 4.2.
  The black node on fig. 4.2a could be of any degree strictly superior to two. If it is within the path that ends at the red node, that path needs to be split. It must be noted that the black path on the left of fig. 4.2a is not valid, due to the degree of the black node that is strictly superior to two. A `split` operation was not applied earlier as it was not required. The `merge` operation depicted on fig. 4.2b must be done if the black node is not a client and is of degree two after the removal of the branch.



(a) Split

(b) Merge

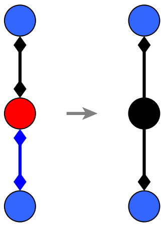
Nodes:

- A **blue** node is either a client or a node of degree > 2
- The **black** node is where the branch of the removed node connects
- The **red** node is the one that is removed
- The blank node is not in the tree any more

Edges: the diamonds represent the ends of a path in the `PathQueue`. Each path has a different colour.

Figure 4.2: `split` and `merge` upon the removal of a degree one client

- *Degree 2*: this case is simpler, as the node should not be removed from the tree. Figure 4.3 shows the `merge` that must be done when such a client is removed.



Nodes:

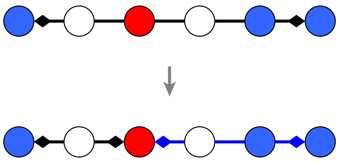
- A **blue** node is a client or of deg > 2
- The **red** node is the client that is removed
- A **black** node is the client after removal, still in the tree

Edges: the diamonds represent the ends of a path in the `PathQueue`. Each path has a different colour.

Figure 4.3: `merge` upon removal of a degree two client

Removing a client of degree strictly higher to two does not require any change in the `PathQueue` because the node stays in the tree, and because of its degree, a `merge` is not possible.

During an improve : the path selected from the `PathQueue` for removal can be invalid. Such a path must be split, and another path must be chosen. As described in fig. 4.4, the path is split around the first invalidating node found in the path.



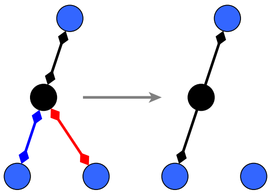
Nodes:

- A **blue** node is should not be within a path, only at its ends
- The **red** node is the one invalidating the path
- A blank node should be within a path

Edges: the diamonds represent the ends of a path in the `PathQueue`. Each path has a different colour.

Figure 4.4: Path split when a chosen path is found to be invalid during an improve

When the selected path is valid and removed, we might need to `merge` two paths around the first node of the selected path. This situation is illustrated on fig. 4.5.



Nodes:

- A **blue** is a client or of deg > 2
- The **black** is not a client and becomes of degree 2

Edges: the diamonds represent the ends of a path in the `PathQueue`. Each path has a different colour, the **red** path is the one that is removed.

Figure 4.5: Paths merge upon the removal of a path during an improve

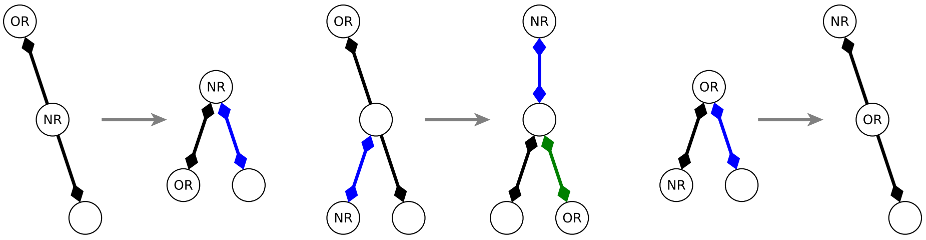
Upon a `reroot` of the downstream component : as the degree of the root before rerooting (*old root*: OR) and the degree of the root after rerooting (*new root*: NR) change due to an improve, the paths around those nodes must be updated. The paths might also be need to be updated because the directions of the edges in the rerooted tree change. The three possible cases shown on fig. 4.6 are:

Case 1: The new root was within a path
Case 2: Inverting the edges might cause the need to `split` paths
Case 3: The paths joining at the old root might need to be merged

For the operations on the `PathQueue` to be efficient, we maintain two dictionaries:

`parentPaths`: its keys are nodes of the tree and the corresponding values are paths in the `PathQueue`. For each key in the dictionary, the associated path is the one that ends at the key.

`childrenPaths`: its keys are nodes of the tree and the corresponding values are lists of paths from the `PathQueue`. For each key, the associated paths are the ones that start with the key.



(a) Case 1: Split **(b)** Case 2: Split **(c)** Case 3: Merge

Nodes:

- OR is the *old root*, the root before rerooting
- NR is the *new root*, the root after rerooting
- A blank node is a node in the tree

Edges: the diamonds represent the ends of a path in the `PathQueue`. Each path has a different colour.

Figure 4.6: `split` and `merge` operations performed upon a `reroot`

Those structures help in matters such as knowing if a node is at an end of a path or within it. They need to be updated at the same time as the `PathQueue`. We want them to stay consistent with the `PathQueue` at all time.

4.2.4 File formats

We conformed our implementation to some file formats, for it to be more easily compatible with other software and contexts.

Some additional file formats were defined to support functionalities from the algorithm, such as for keeping in sync the pre-computed shortest paths and the topologies.

`GML` file format

The on-file graph representation we chose to use is the `GML`. This file format is widely used. In it, nodes and edges can receive attributes which are in sync with what `NetworkX` supports.

The attributes we support are the following: `WEIGHT, GEO, NONE`. The `network graph` data structure, dealing with those attributes, is described in section 4.2.1.

`NetworkX` is able to import `GML` files natively.

However, not all topologies are encoded using the `GML` format. In our evaluation, we used topologies from different sources. They are often encoded using other file formats : `NTF`, `GraphML` and `INTRA`. We wrote scripts to convert those topologies to the `GML` format.

Topology file formats

A configuration file format for defining topologies was created. It aims at linking `GML` topology files along with their shortest paths binary files. Those shortest paths are the result of the pre-computation phase done on the network before trees are actually built. Topology files also store the attribute on which to base the weights used in the calculations.

Scenario files

A scenario file format for defining scenarios, which are then used in our evaluation in chapter 5, has also been created. A file of this format contains events composing the scenarios. A scenario file can then be input to one of the testing algorithms that we wrote. All those concepts are described further in the document.

4.3 Side algorithms

We present here how we handled several technical issues, such as the computation of edge weights in a certain scenario, the computation of shortest paths, and how we handle the temperature value for the simulated annealing is evaluated based on time.

4.3.1 Haversine distance

When the used attribute is geographical coordinates (`GEO` parameter), link weights are derived from latitude and longitude coordinates of nodes in the the network. The geographical distance between nodes at the ends of each link is used as its `weight` attribute.

HAVERSINE ALGORITHM

The *haversine algorithm* computes a realistic distance using trigonometric relations, taking into account the radius of the Earth.

The implementation we used is based on the idea of Wayne Dyck [34].

4.3.2 Yen's algorithm and shortest paths computations

As it was said earlier, the shortest paths between all the nodes of the graph are pre-computed. We initially wanted to compute the k-shortest paths for each pair of nodes in the graph, so that we would have more choice regarding reconnection paths when reconnecting the components created during an improvement step.

Doing so would have added some diversity to the search. In the end, we did not use that feature, but implemented everything so that those paths could be computed. Here, we set the value of k to 1, which amounts to finding a single shortest path between each two nodes in the graph.

The implemented algorithm is the *Yen's algorithm* [35]. It computes the k-shortest paths between two nodes in a graph. The algorithm uses *Dijkstra's algorithm* [27] to compute the shortest path between the two nodes, and then goes through the edges of the path, removing them one by one and trying to find new shortest paths not using the removed edges.

The implementation of Dijkstra's algorithm we use is the one provided in the library `NetworkX`.

To compute all our shortest paths, for each node in the graph, we have to try and reach all the other nodes in the graph. As the links of the graphs we used are bidirectional, a shortest path computed from a node u to a node v is the same as the one from v to u, except it is reversed. This is also true for the k-shortest paths. We took that property into account when computing the shortest paths, in order to avoid computing each path twice.

4.3.3 Simulated annealing

In order to apply the simulated annealing meta-heuristic described in section 3.3.7, we need to define how temperature should behave. If the `temperature schedule` parameter (cf. section 3.4) is set to `CONSTANT`, there is no need to re-evaluate it over time. It is set to an arbitrary value at the beginning of the execution.
As explained in section 3.3.9, the probability to degrade is based on the temperature, on the weights of the path that was removed and on the degrading path chosen to replace it.

$$p = e^{\frac{-\Delta w}{temp}}, \quad \Delta w = 100 \times \frac{weight(ReplacementPath) - weight(RemovedPath)}{weight(ReplacementPath)}$$

We set the arbitrary value to 10 so that we have the following probabilities:

Δw [%]	50	37.5	25	20	12.5	10	7.4	5.88
p [%]	0.67	2.35	8.2	13.5	28.6	36.8	47.7	55.5

On the contrary, if it is set to `LINEAR`, we use the elapsed time since the beginning of the improvement step to determine the temperature for each call to `ImproveOnce`. Temperature value *temp* is evaluated as such, *maxTime* being the time allocated for the improvement step:

$$temp = maxTime - elapsedTime$$

In our implementation, that *temp* value is divided by 10 so that it fits with our probability evaluation.

To evaluate *elapsedTime*, we use the `clock()` function from Python's `time` module. We used `clock()` over `time()` because the latter measures `CPU time` instead of wall time. According to the Python documentation, `clock` is the right function to use for benchmarking small portions of code and timing algorithms [36].

Chapter 5

Evaluation and testing

We evaluate our implementation through testing. First, we describe the conditions in which tests were run, as well as the metrics used for assessing about tree quality over the course of an execution.

In a second time, we present which experiments were conducted and their results.

5.1 Testing conditions

The testing conditions under which our tests were run are described in four steps.

First, we describe how we built the scenarios processed by our algorithm. Scenarios are composed of events, namely `join` and `leave` events from nodes in a certain topology, occurring between `ticks`, which represent iterations from an arbitrary clock. Those scenarios constitute our datasets.

Then, we briefly explain how we scheduled improvement steps in our experiments.

In a third step, we describe which metrics were used to evaluate the quality of the built trees and the performance level of the algorithm.

Finally, we characterise the topologies we used.

5.1.1 Scenario generation

In order to test the construction of multicast trees, we had to simulate what multicast events (`join` and `leave` events of subscribers) could happen on a specific topology.

Scenarios based on real data that go along topologies are hard to get. They were therefore generated probabilistically following a characterisation of node behaviour. This probabilistic way of generating scenarios aims at simulating realistic events.

To generate our lists of events, we associate a couple ($\mathbb{P}(\texttt{join})$, $t(\texttt{tree})$) to each node in the graph, where $\mathbb{P}(\texttt{join})$ is the probability for the node to join the tree at each `tick`, and $t(\texttt{tree})$ the average time such a node stays in the tree. We chose to use the same couple across all nodes in the topology, which is not a limitation because the model is probabilistic.

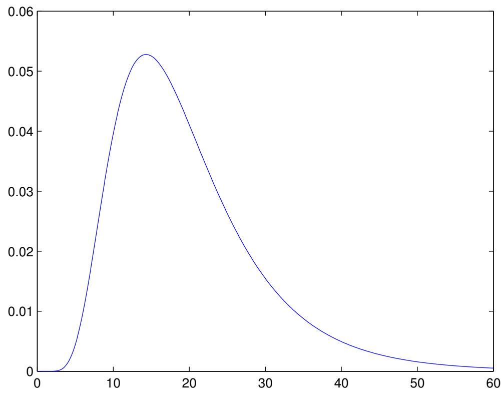
We use a discrete time basis to compute the events lists. Each generated list has `ticks` times in it. The generation iterates `ticks` times and populates the scenario with join and leaves events.

At each discrete step S_i of the generation, according to the value given to $\mathbb{P}(\texttt{join})$, a certain amount of nodes will join the multicast group. For each of those new subscribers, we add a `join` event at step S_i. We also evaluate how much time the node will stay in the tree, that is, at which step a `leave` event should be added to the scenario. The amount of time t the node stays in the tree (expressed as a number of discrete steps) is based on the value given to $t(\texttt{tree})$ for this node. To find a specific time around the value given to $t(\texttt{tree})$, we follow a normal distribution $\mathcal{N}(m, sd)$ with a specific mean and standard deviation. As normal distribution can lead to values below zero, we convert the normal distribution to a log-normal one, $\ln \mathcal{N}(\mu, \sigma)$, which does not yield negative values. Indeed, a negative time has no meaning in this context.

We use the following conversion scheme:

$$\mu = \ln\left(\frac{m^2}{\sqrt{v + m^2}}\right), \quad \sigma = \sqrt{\ln\left(1 + \frac{v}{m^2}\right)} \quad \text{with } v = sd^2$$

We only provide a value for mean m, this is the value $t(\texttt{tree})$. For the standard deviation, we arbitrarily use value $sd = \frac{m}{2}$. The shape of the distribution is given at fig. 5.1.



Using this distribution yields a large range of values for the time to stay in the tree. The following table gives confidence intervals at 95% for several values of m:

m	lb	ub
10	3.54	22.58
50	17.72	112.88
150	53.15	338.64

There is a linear relation between the mean and the bounds of the confidence interval, the table gives some examples.

Figure 5.1: $\ln \mathcal{N}(\mu, \sigma)$ derived from $\mathcal{N}(20, 10)$

When time to stay t in the tree for node is evaluated, we add a `leave` event for that node at step S_{i+t}. Of course, for each step, only nodes not already in the tree are considered as candidates to join. Once all potential nodes have been considered at step S_i, the next step S_{i+1} is evaluated.

Once the iterations are over, the `ticks` first steps are extracted from the produced list of events. Events at one step occur during the period between the previous and the next `tick`.

We generated three kinds of scenarios, each with varying values for $\mathbb{P}(\texttt{join})$ and $t(\texttt{tree})$, in order to "discretise" the broad range of node behaviours that can potentially happen in a network. The scenarios put our algorithm under different levels of "stress". For each level of stress and each topology (described in section 5.1.2), three scenarios were generated. Characteristics about each of those scenarios are summarised in table 5.1. The columns *clients* in the table are the number of clients in the tree when all the events have been applied.

Table 5.1: Summary of scenarios characteristics

Stress level	$\mathbb{P}(\texttt{join})$	$t(\texttt{tree})$	ticks	`Tiscali`		`KDL`		`ANON1`	
				events/`tick`	clients	events/`tick`	clients	events/`tick`	clients
medium	0.05	25	100	6.62	89	32.16	433	27.59	378
				6.79	90	32.59	432	27.34	347
				6.91	98	31.93	398	27.4	335
stable	0.2	50	100	5.93	148	27.92	695	24.44	587
				6.08	151	27.59	684	24.5	583
				6.24	149	28.35	690	23.65	590
unstable	0.2	10	100	21.28	111	99.52	507	84.69	436
				20.8	113	98.59	522	84.48	429
				21.28	101	98.99	530	84.27	436

5.1.2 Topologies

In our tests, we use several topologies.

The first one is `Tiscali`, which is topology number 3257 from `RocketFuel` [37] database [38]. In this topology, a weight and latency are associated to each link. We only use the weight attribute. It is encoded with `.intra` files that we converted to the `GML` format for them to be understandable by our implementation. The topology contains 161 nodes and 328 edges. Because it contains twice as many edges as it has nodes, the search space for finding multicast trees is large with respect to the the number of nodes, if the graph had fewer edges, there would be less possible multicast trees. As our goal is to try and improve by selecting improving path throughout the construction of the tree, this topology is a good choice.

The second topology we used is `KDL` (Kentucky Datalink) from the *Internet Topology Zoo*[39], which contains 754 nodes and 899 edges. We used it for measuring the time needed to add and remove clients from the tree. Due to the high number of nodes in it, it allows us to experiment with a large topology. We also used it to measure the impact on the network when performing improvement steps. We had to compute the weights based on the geographic locations of the nodes.

The third topology, which we name ANON1, contains about 600 nodes and 1000 edges. We use it when measuring the impact of the improvement steps on the network. It has many nodes like KDL, but has in average more edges per node, more alternative paths are thus exploitable. The weights were available in the topology file.

Table 5.2 summarises the information about the three topologies.

Table 5.2: Summary of the different topologies that were used in our tests

Topology	Nodes	Edges	Weight attribute	Origin
Tiscali	161	328	WEIGHT	RocketFuel
KDL (Kentucky Datalink)	754	899	GEO	Topology Zoo
ANON1	∼600	∼1000	WEIGHT	n/a

During the development of our solution, we also used two other topologies which are listed in table 5.3.

Table 5.3: Topologies we used during the development

Topology	Nodes	Edges	Weight attribute	Origin
Belnet2010	22	32	unit cost	Topology Zoo
Cogent (Cogentco)	196	245	GEO	Topology Zoo

5.1.3 Scheduling of improvement steps

improvement steps

Aside from join and leave events, we can also perform *improvement steps*. They are not to be added to scenarios, as they do not characterise the behaviour of network nodes. Instead, they should be independently scheduled by network administrators, so as to react to network events.

Implementation-wise, improvement steps are processed as regular events. Each improvement event has for argument the amount of time allocated for this improvement. In our tests, a single parameter, namely, the maximum improve time, specifies the time allocated to each improvement step.

Where to add them in the list depends on another parameter, namely, the improve period, defined in section 3.4. It states the period – in terms of ticks – at which to schedule the improves. As an example, a period of 1 would schedule an improve event after each tick.

An example of such a scenario is given on fig. 5.2. This scenario is made of 10 ticks. Events e_1 to e_{14} are the events occurring between tick 4 and tick 5.

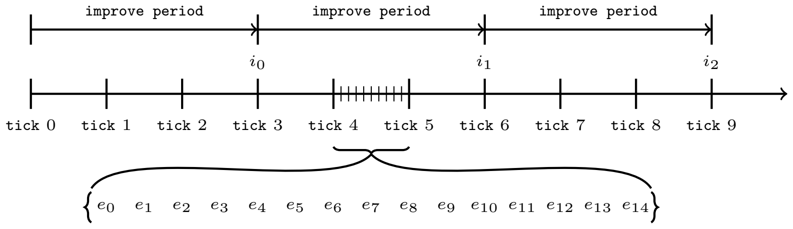


Figure 5.2: Scenario structure made of 10 `ticks`

In the example, the `improve period` is set to three, meaning that an improve is performed every three `ticks`.

5.1.4 Metrics

To measure the quality of the tree built by our method and the performance of the algorithm, we define several metrics.

The `averaged integral` metric

Our first guess was to use final (after all events have been processed) tree weights as a metric to compare executions and assess about tree quality. However, using this simplistic metric, we are not able to assess about tree quality when such a tree is being built.

Instead, we want to make sure that the tree is competitive throughout its building process. We want our solution to be competitive in an online environment, that is, the algorithm should output good trees whether we are at the beginning or at the end of a simulation.

We therefore take into account in our metrics the weight of the tree at each step of its construction.

We therefore evaluate the trees we are building based on the discrete steps that were used during the generation of the events lists, that is, the *Ticks* `ticks` "*clocking*" the scenario.

AVERAGED INTEGRAL

The first metric we used is an *averaged integral.* After each one of the `ticks` of the scenario, the weight of the tree is calculated and logged. Once the scenario finishes, the gathered tree weights are averaged and this average value is logged.

This metric then gives the average weight of the tree at the end of each step.

Our goal is to minimise the value of this metric.

It yields a good tool for comparing different executions running on the same scenarios, that is, different configurations of the algorithm, for building distribution trees for a common set of clients.

Impact on nodes

IMPACT The second metric we use serves as a way to measure the *impact* that improvement steps have on the global network. For this metric, we identify which nodes are impacted in the graph for each step of improvement.

This metric gives valuable information as every solution that is to be implemented in an `OpenFlow` network should consider the interactions needed between the controller and network switches. This metric gives us information on the number of devices whose flow tables must be updated after an improvement step. It enables us to reason about when and for how long improvement steps should be scheduled.

We express this impact as the proportion of nodes from the topology directly impacted by the improvement, that is, which switches should see their flow tables updated.

Group events processing time

Another test that we run was measuring the CPU clock time required for handling multicast group events, namely, `join` and `leave` events. This test is important in order to validate our assumptions on time complexity for the `Greedy` approach that we use for handling such events. In particular, we plot the time needed for handling those two kinds of events with respect to the number of nodes present in the tree at that moment. We then highlight the linear nature of our `Greedy` approach, with respect to this variable.

5.2 Experiments

The experiments that we performed are described in this section. All of the experiments were done on an `Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz` processor, with 8Gb of RAM.

In a first experiment, we compared three methods: our optimised method, the `Greedy` non improved method, that is, using our algorithm without scheduling improvement steps, and the current `PIM-SSM` approach followed in classical networks. The `PIM-SSM` protocol has been described in section 1.1.3.

A second experiment then assesses whether our method can be used in a real environment by depicting the time taken for handling multicast group change events, namely, `join` and `leave` events.

Then, we motivate the choice of parameters we made for the optimised method by determining the set of parameters that works best as well as determine whether it is important to spend much time on such improving steps.

Finally, we measure the impact that improvement steps have on the network, that is, how many switches from the topology are affected by improvement steps.

For all experiments, if not stated otherwise, the topology that was used is Tiscali.

5.2.1 Optimisation versus Greedy versus PIM-SSM

PIM-SSM is the current state of the art for achieving multicast in classical IP networks. We want to confront this classical approach to a basic algorithm based on the global knowledge brought by the SDN approach, namely the Greedy approach, and to our optimised way of building the tree by scheduling additional improvement steps. For this experiment, we used the averaged integral metric, defined in section 5.1.4. The experiment was done using the Tiscali topology.

COMPARISON TABLES

comparison tables are based on the averaged integral metric. In such table, each column represents a different configuration of the algorithm, while each of the scenarios is laid out horizontally.
For each combination (*scenario*, $configuration$), the algorithm is run several times and the averaged integral values are collected.
The average on the averaged integral values of the several runs is then computed and used as entries in the tables. Running each combination several times is done so that we take into account the non-deterministic nature of the algorithm. In some cases, the algorithm is fully deterministic so we can run it only once.

We then perform several operations on the table, in order to be able to compare the performance of the different configurations across the different scenarios.

First, we normalise the values in the table horizontally, according to a reference column. That is, the value in the reference column is scaled to 1.0, while values from the other columns are scaled by the same factor. The reference column can either be predefined, so as to be able to compare the performance of a specific configuration relative to the others, or determined automatically to the minimum value across the columns for each line, so as to compare the best performing configuration relative to the others.

When a table contains too many columns, it is split in several smaller tables, the reference column is then present in only one of the sub-tables.

In a second and last step, a geometric mean is computed across the rows, on a column basis. This geometric mean is displayed in the last row of the tables. The geometric mean of configuration j is computed as such:

$$\sqrt[m]{\prod_{i=1}^{m} norm_{j,i}}$$

The structure of the `comparison tables` is given in table 5.4.

Table 5.4: Example of `comparison table` for m different scenarios and n configurations. $norm_{i,j}$ are already normalised according to the reference column. The table shows the formulas behind the computation of the geometric mean.

	Configuration 1	...	Configuration k	...	Configuration n
Scenario 1	$norm_{1,1}$	...	$norm_{k,1}$	...	$norm_{n,1}$
...	...	...	...	...	...
Scenario j	$norm_{1,j}$	...	$norm_{k,j}$	...	$norm_{n,j}$
...	...	...	...	...	...
Scenario m	$norm_{1,m}$	...	$norm_{k,m}$	...	$norm_{n,m}$
Geometric mean	$\sqrt[m]{\prod_{i=1}^{m} norm_{1,i}}$	...	$\sqrt[m]{\prod_{i=1}^{m} norm_{k,i}}$	...	$\sqrt[m]{\prod_{i=1}^{m} norm_{n,i}}$

The value in the last row then gives us insight about how a configuration performs relative to the others, across the different scenarios.

In such tables, we keep three significant figures. The value precision goes up to the tenth of percent. We do not want to limit our precision to a percent basis.

An simple example of such tables is given at fig. 5.3.

	`cfg1`	`cfg2`	`cfg3`
Scenario 1	10	13	16
Scenario 2	15	12	18

Example values

	`cfg1`	`cfg2`	`cfg3`
Scenario 1	1.0	1.3	1.6
Scenario 2	1.25	1.0	1.5
Geo. mean	**1.118**	**1.14**	**1.549**

Comparison without specific reference column

	`cfg1`	`cfg2`	`cfg3`
Scenario 1	0.768	1.0	1.231
Scenario 2	1.25	1.0	1.5
Geo. mean	**0.981**	**1.0**	**1.359**

Comparison with `cfg2` as reference

This simple example presents what `comparison tables` look like.
The first table shows made-up values for the mean of `averaged integral` values for two scenarios and three configurations.
The second table shows the normalisation of the values for each scenario. The reference column can change between scenarios. It corresponds to the smallest value for a given scenario. The last line is the geometric mean of the values in each column. The results highlights that the first configuration performs better, as the associated geometric mean is lower than the others.
The third table uses `cfg2` as a reference. Each value across the columns is then normalised with respect to the value in the second column. The geometric means are then computed. This table shows that `cfg1` is better than `cfg2` as its geometric mean is smaller. The third configuration performs worst. In this last table, the geometric mean of the `averaged integral` values of `cfg3` are 35% higher than those of `cfg2`.

Figure 5.3: Simple example of `comparison tables`

Here, we want to compare three configurations. The first configuration of our algorithm used is our optimised method of computing multicast trees. The second one follows the same configuration except that improvement steps are not scheduled, which amounts to following a `Greedy` approach. The third and last configuration corresponds to the classical `PIM-SSM` approach.

This test highlights the benefits brought by the `SDN` approach and the additional gain brought by our improving method, run on top of the `Greedy` method.

Table 5.5: Optimisation versus `Greedy` versus `PIM-SSM`

Configuration:	Optimisation	Greedy	PIM-SSM
medium	0.708	0.816	*1.0*
	0.723	0.799	*1.0*
	0.742	0.789	*1.0*
stable	0.713	0.778	*1.0*
	0.714	0.777	*1.0*
	0.711	0.781	*1.0*
unstable	0.722	0.79	*1.0*
	0.723	0.782	*1.0*
	0.717	0.766	*1.0*
Geometric mean	**0.719**	**0.786**	***1.0***

As shown on table 5.5, having a complete knowledge of the network allows us to build trees that are less costly even when no mechanism is used for improving. This fact is highlighted by comparing the `averaged integral` values and geometric means of such values between the `Greedy` and the `PIM-SSM` approaches.
We see that improving the tree further reduces the `averaged integral` values of the trees considered when improvement steps are scheduled throughout the building process. This fact is highlighted by the differences between the Optimisation and `Greedy` columns.

The default optimised configuration that was used in order to generate the `averaged integral` table is depicted in fig. 5.4.

```
    maximum improve time = 25,
          improve period = 1,
    improve search space = ∞,
          intensify-only = False,
  maximum selected paths = 1 (only with MOST EXPENSIVE PATH),
         search strategy = BEST IMPROVEMENT,
edge selection heuristic = MOST EXPENSIVE EDGE,
       tabu time-to-live = 50,
    temperature schedule = LINEAR
```

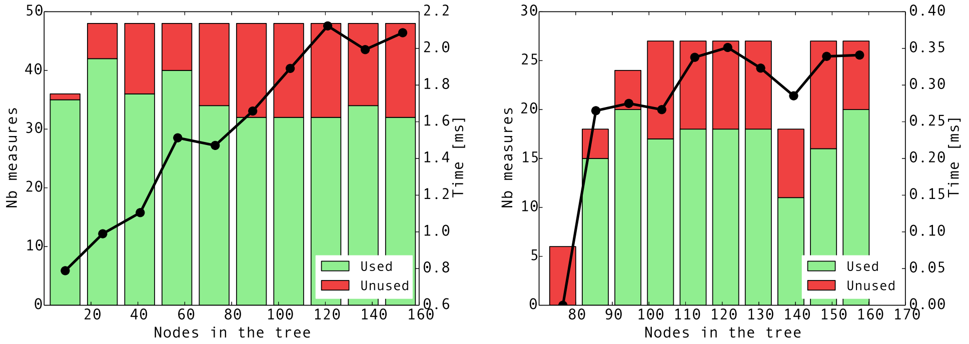
Figure 5.4: Default optimised configuration

5.2.2 Time for adding and removing clients

Adding and removing a client must be done quickly when achieving multicast. There are several sources of delay besides the computation time required for handling events, such as the update of switches' flow tables and the delay on the links.

We want our method to perform those actions as quickly as possible in order to minimise the total delay. The following graphs shows the time taken for the addition and removal of client, relative to the number of nodes present in the tree at the time the event is processed. As it is possible for a subscribing node to be already in the tree, some additions do not incur any modification to the tree and take a negligible amount of time. For our tests to be meaningful, the addition times required for adding a node already in the tree were not taken into account. Indeed, the process of adding a client already in the tree is done in constant time.

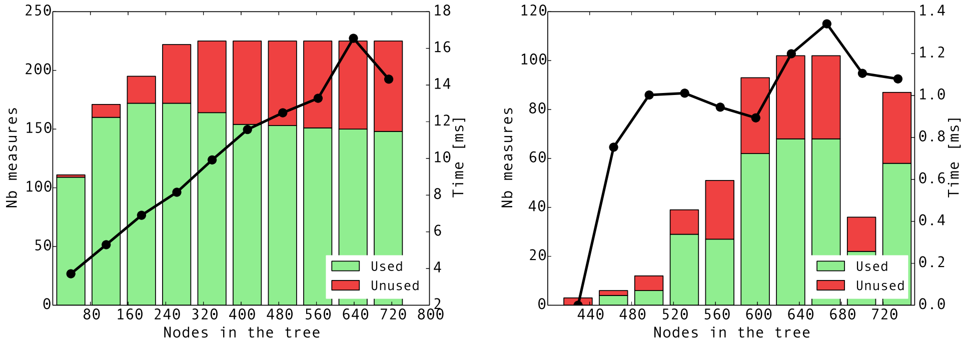
Similarly, upon the removal of a client, the tree is only modified when the client is a leaf of the tree. Only the times for removing a degree one client were kept. The test thus gives us an upper bound on the time required for handling such events.



(a) Addition times

(b) Removal times

Figure 5.5: Addition and removal times on `Tiscali` topology



(a) Addition times

(b) Removal times

Figure 5.6: Addition and removal times on `KDL` topology

On fig. 5.6, the horizontal axis represents the number of nodes in the tree when the additions or removals occur. The left vertical axis represents the number of measures per histogram bucket, and the right one is related to the curve and represents the average time needed to process an event. The measures that are displayed in red on the bar charts are the ones that were not used to compute the average time to add or remove a client.

On those graphs, we can validate the assumption relative to time complexity of adding a client that was given in chapter 3. We notice that the time needed for the addition of clients is proportional to the number of nodes in the tree.

When it comes to removing nodes, we do not clearly see the linear time complexity of the operation, because in practice, the removal of a node of degree one only involves the removal of a few edges of the tree. The worst case scenario, which corresponds to removing all the edges of the tree, only occurs when removing the last and only client of the tree. The best case scenario when removing a degree-one client is removing one edge.

As seen on the figures, applying an event to the tree only takes from a fraction of millisecond for a `leave` event and up to a few milliseconds for a `join` event.

The time needed to perform the addition and removal of clients is thus not a limiting factor for practical use in a network.

5.2.3 Choice of values for the parameters of the improvement method

Changing the parameters of the algorithm might have an impact on the solution.

In this section, aside from the `edge selection heuristic` parameter, we treat parameters separately in order to motivate the choice of value made for each of them in section 5.2.1. Each of the implemented heuristics for edge selection was used for every parameter, except when such parameter was only applicable to one specific heuristic. We can then motivate a choice of such an heuristic on an enlarged range of tests.

Because scheduling improvement steps yields a ten percent reduction in `averaged integral` cost with respect to the `Greedy` approach, an improvement of a few percents from one value of a parameter to another value is meaningful.

Each of those parameters influence the global behaviour of the search algorithm. The contribution of a single argument to the global level of performance is often reduced. However, the small increase in performance brought by optimising such arguments can yield a significantly greater level of performance when such optimisations are combined together. It thus makes sense to try to optimise the arguments in order to single out values that perform best for each of them.

In order to compare different configurations, corresponding to different values of each parameter, we use the `averaged integral` metric, as defined in section 5.1.4, in `comparison tables`. For each test made, the performance of the `Greedy` algorithm is used as point of comparison, which is known to perform better than `PIM-SSM`. Our goal is here to optimise the improvement method applied on top of the `Greedy` tree construction.

When one of the configuration corresponds to the default optimised configuration defined in fig. 5.4, it is set as the reference for the experiment.

`maximum improve time & improve period`

The `maximum improve time` and `improve period` parameters must be considered together, as they define the total amount of time that is spent on improvements throughout the execution.
Indeed, scheduling ten improvement steps, each of them running for one second, or improving one hundred times, each one for a hundred milliseconds, gives the same total amount of time spent on improving the tree.

CONFIGURATION FAIRNESS

In order to keep this level of "*configuration fairness*" in terms of total time allocated for improving between algorithm configurations, the two arguments should be considered together. When scheduling improvement steps differently, we make sure to keep the ratio `maximum improve time`/`tick` constant. We arbitrarily decided to set this ratio to 25 ms per `tick` event. As an example, if improvement steps are scheduled every five `ticks`, each of those improvement steps will be allocated 5 × 25 ms = 125 ms.

Table 5.6: Influence of the `maximum improve time` and `improve period` parameters

	Edge selection: `RANDOM`					`Greedy`
`ip/it:`	**1/25**	**5/125**	**10/250**	**25/625**	**50/1250**	
medium	1.003	1.004	1.019	1.054	1.09	*1.155*
	0.998	0.993	1.003	1.05	1.074	*1.102*
	0.991	0.991	1.0	1.029	1.043	*1.063*
stable	1.0	1.003	1.013	1.025	1.047	*1.092*
	1.007	1.007	1.015	1.03	1.05	*1.089*
	1.007	1.008	1.015	1.033	1.06	*1.1*
unstable	1.009	1.01	1.02	1.041	1.061	*1.094*
	1.01	1.015	1.025	1.042	1.054	*1.082*
	1.008	1.011	1.022	1.036	1.044	*1.07*
Geometric mean	**1.004**	**1.005**	**1.015**	**1.038**	**1.058**	***1.094***

Table 5.7: Influence of the `maximum improve time` and `improve period` parameters

	Edge selection: `MOST EXPENSIVE PATH`					`Greedy`
`ip/it:`	**1/25**	**5/125**	**10/250**	**25/625**	**50/1250**	
medium	0.997	0.998	1.015	1.054	1.089	*1.155*
	0.99	0.989	1.004	1.045	1.072	*1.102*
	0.992	0.986	0.997	1.03	1.043	*1.063*
stable	0.999	0.995	1.001	1.017	1.043	*1.092*
	1.01	1.0	1.006	1.023	1.047	*1.089*
	1.016	1.002	1.008	1.026	1.058	*1.1*
unstable	0.996	0.998	1.012	1.039	1.06	*1.094*
	1.004	1.003	1.019	1.038	1.054	*1.082*
	0.999	1.005	1.017	1.032	1.045	*1.07*
Geometric mean	**1.0**	**0.997**	**1.009**	**1.034**	**1.057**	***1.094***

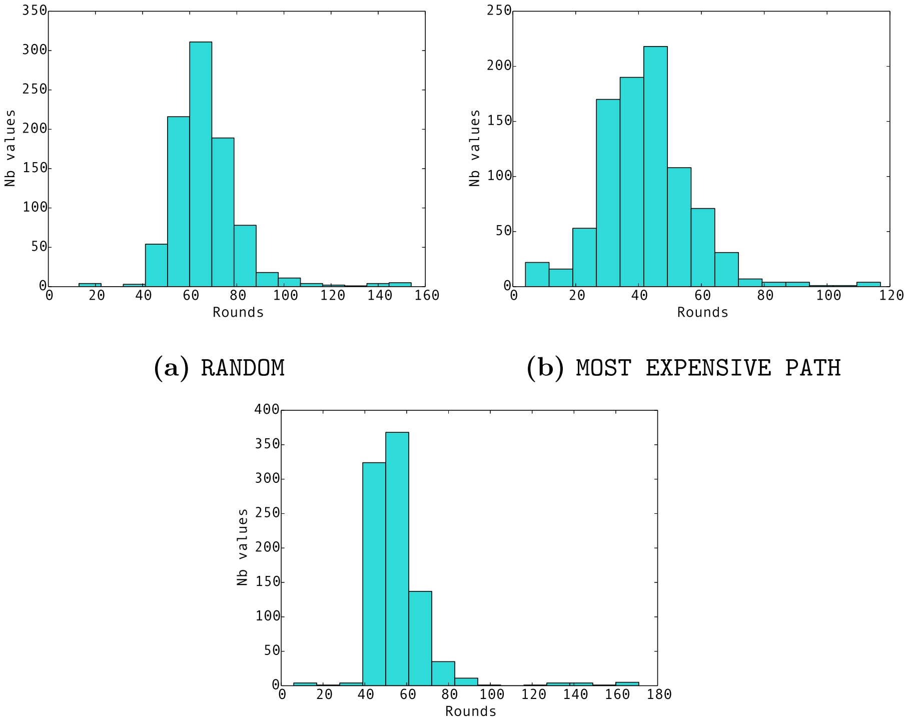
Table 5.8: Influence of the `maximum improve time` and `improve period` parameters

	Edge selection: `MOST EXPENSIVE EDGE`					`Greedy`
`ip/it:`	**1/25**	**5/125**	**10/250**	**25/625**	**50/1250**	
medium	1.0	1.001	1.013	1.05	1.087	*1.155*
	1.0	0.991	1.004	1.047	1.073	*1.102*
	1.0	0.988	0.998	1.026	1.041	*1.063*
stable	1.0	0.995	1.001	1.018	1.044	*1.092*
	1.0	1.002	1.007	1.023	1.046	*1.089*
	1.0	1.002	1.008	1.025	1.056	*1.1*
unstable	1.0	0.999	1.012	1.038	1.059	*1.094*
	1.0	1.005	1.021	1.039	1.053	*1.082*
	1.0	1.008	1.018	1.032	1.044	*1.07*
Geometric mean	**1.0**	**0.999**	**1.009**	**1.033**	**1.056**	***1.094***

The `comparison tables` in tables 5.6 to 5.8 are related to each other. Their reference column is set on the first column of table 5.8.

The results show that it is beneficial to improve often. Indeed, with an `improve period` of 50, the solution is 5% worse compared to the reference configuration.

Despite the difference of 5% between the best and worst choice for `improve period` and `maximum improve time`, the solution is still 5% to 10% better than the `Greedy` one.



(a) `RANDOM`

(b) `MOST EXPENSIVE PATH`

(c) `MOST EXPENSIVE EDGE`

Figure 5.7: Rounds of local search per heuristics for `ip/it` = 5/125

The reason is that the simpler heuristics (`RANDOM` and `MOST EXPENSIVE EDGE`) can perform more rounds of local search during an improvement step. The number of rounds is depicted on fig. 5.7.

The `MOST EXPENSIVE EDGE` selection heuristic does slightly fewer rounds than `RANDOM` selection because the most expensive edge must be selected by looking at all the edges of the tree. When using the `MOST EXPENSIVE PATH` selection heuristic, less rounds are made than with the two other heuristics. This is due to the complexity of maintaining the `PathQueue` data structure.

The conclusion we can draw is that `MOST EXPENSIVE PATH` is more efficient if evaluated on a number of local search round basis, but the time needed to execute one round is such that the heuristic equals the others in performance.

`tabu time-to-live`

Here, we want to measure the impact of putting the edges recently added to the tree in the `tabu list`. This concept of `tabu list` and its effect on the local search algorithm is introduced in section 3.3.9.

Table 5.9: Influence of the `tabu time-to-live` parameter

	Edge selection: `RANDOM`						`Greedy`
Time-to-live:	**0**	**3**	**10**	**25**	**50**	**250**	
medium	1.001	1.003	1.002	0.999	0.998	1.0	*1.152*
	1.001	0.999	0.997	0.998	0.998	0.997	*1.104*
	0.997	0.995	0.994	0.995	0.994	0.992	*1.064*
stable	1.002	1.003	0.999	1.001	1.004	1.001	*1.092*
	1.009	1.005	1.006	1.008	1.007	1.007	*1.089*
	1.008	1.016	1.006	1.01	1.008	1.01	*1.1*
unstable	1.012	1.007	1.01	1.008	1.012	1.011	*1.094*
	1.013	1.013	1.013	1.009	1.01	1.011	*1.082*
	1.007	1.009	1.006	1.008	1.009	1.007	*1.068*
Geometric mean	**1.006**	**1.005**	**1.004**	**1.004**	**1.004**	**1.004**	***1.093***

Table 5.10: Influence of the `tabu time-to-live` parameter

	Edge selection: `MOST EXPENSIVE PATH`						`Greedy`
Time-to-live:	**0**	**3**	**10**	**25**	**50**	**250**	
medium	1.029	1.007	0.993	0.993	0.992	0.993	*1.152*
	1.021	1.005	0.991	0.992	0.992	0.991	*1.104*
	1.028	1.005	0.989	0.991	0.988	0.988	*1.064*
stable	1.017	1.008	1.004	0.997	0.999	0.997	*1.092*
	1.016	1.016	1.007	1.01	1.009	1.008	*1.089*
	1.021	1.014	1.008	1.015	1.01	1.015	*1.1*
unstable	1.008	1.0	0.995	0.996	0.996	0.995	*1.094*
	1.021	1.004	1.002	1.002	1.002	1.004	*1.082*
	1.014	1.001	0.995	0.996	0.997	0.997	*1.068*
Geometric mean	**1.02**	**1.007**	**0.998**	**0.999**	**0.998**	**0.999**	***1.093***

Similarly to previous section, tables 5.9 to 5.11 are to be considered together. The reference column across the tables is the fifth one of table 5.11.

We can draw the conclusion that the `ttl` value has no influence when using the

Table 5.11: Influence of the `tabu time-to-live` parameter

	Edge selection: `MOST EXPENSIVE EDGE`						`Greedy`
Time-to-live:	**0**	**3**	**10**	**25**	**50**	**250**	
medium	1.037	1.025	1.0	0.999	1.0	0.996	*1.152*
	1.031	1.025	0.999	0.997	1.0	0.999	*1.104*
	1.037	1.034	1.003	1.001	1.0	1.001	*1.064*
stable	1.02	1.018	1.003	0.999	1.0	0.999	*1.092*
	1.025	1.019	1.0	0.999	1.0	0.999	*1.089*
	1.034	1.025	1.001	1.0	1.0	1.001	*1.1*
unstable	1.027	1.021	1.003	1.0	1.0	1.0	*1.094*
	1.026	1.021	1.001	1.0	1.0	1.0	*1.082*
	1.034	1.028	1.0	0.998	1.0	0.998	*1.068*
Geometric mean	**1.03**	**1.024**	**1.001**	**0.999**	**1.0**	**0.999**	***1.093***

`RANDOM` edge selection heuristic. Indeed, the `tabu list` does not help, as `RANDOM` is non-deterministic by nature.

When using the `MOST EXPENSIVE PATH` selection heuristic, we notice that the `ttl` must be higher than zero. Regarding `MOST EXPENSIVE EDGE`, a slightly larger value of `ttl` is needed for the heuristic to perform well.

This is expected as the two latter selection heuristics order the edges or paths from the tree in a deterministic manner. Without a `tabu list` (case when the `ttl` value is zero), the same edges would be considered over and over, leading to no improvement.

A large enough value for this parameter thus brings diversification to the search, and leads to a better-performing algorithm.

The results show that any large enough value can be chosen. Indeed, setting the `ttl` to a value of 50 or 250 does not significantly improve the performance of the search, nor does it worsen it.

`Intensify only`

As described in section 1.3.3, a local search approach often makes use of a good balance between *intensification* and *diversification*. It is therefore sometimes needed to degrade a solution in order to escape from a local optimum.

The following experiment aims at measuring under different conditions if it is indeed necessary to degrade when building multicast trees.

Setting the `boolean` parameter `intensify-only` to `True` prevents the algorithm from degrading a solution. In such configuration, the temperature used in the simulated annealing approach is of no use as such temperature is only used to evaluate a probability to degrade the tree during the search.

Results of this experiment are depicted on table 5.12. As seen on the previous experiments, the influence of the selection heuristic is really thin here. However, one conclusion that can be drawn is that allowing to degrade during the search does

Table 5.12: Influence of the `intensify-only` parameter.
`True` means that no improvement step is allowed to degrade the tree.

Edge selection:	`RANDOM`		`MOST EXPENSIVE PATH`		`MOST EXPENSIVE EDGE`		`Greedy`
`intensify-only:`	`True`	`False`	`True`	`False`	`True`	`False`	
medium	1.001	1.001	0.998	0.995	0.999	1.0	*1.154*
	1.0	1.002	0.993	0.992	1.0	1.0	*1.107*
	0.997	0.995	0.987	0.989	1.001	1.0	*1.063*
stable	0.999	1.001	1.002	1.003	0.999	1.0	*1.092*
	1.006	1.006	1.01	1.009	0.999	1.0	*1.089*
	1.005	1.005	1.014	1.013	0.999	1.0	*1.099*
unstable	1.006	1.011	0.995	0.995	1.0	1.0	*1.093*
	1.01	1.012	1.003	1.003	1.0	1.0	*1.082*
	1.008	1.009	0.998	0.998	1.001	1.0	*1.069*
Geometric mean	**1.004**	**1.005**	**1.0**	**0.999**	**1.0**	**1.0**	***1.094***

not help finding better solutions, meaning that either there are only a few local optima and that intensification is enough for searching a broad neighbourhood, or that our search is not broad enough to escape from such local optima.

Despite the lack of significance of such results, we could argue that preventing degradations yields slightly better solutions. The reason behind that fact might be that if a degradation is done in an early round of the improvement step, it worsens the solution at the beginning of the search. As such degrading edges are also added to the `tabu list`, they will not be reconsidered until their `ttl` value reaches zero. If the improvement step is not allocated enough time, that is, if it is not able to perform a sufficient amount of rounds of improvement, the search is unable to compensate the degradation over time.

However, the experiment made in section 5.2.3 on the `maximum improve time` and `improve period` showed that it was more interesting to give a smaller amount time more often then the opposite.

`Temperature schedule`

We implemented two possible ways of scheduling the temperature used by the simulated annealing approach. This is chosen via the `temperature schedule` parameter. In this experiment, we want to determine if one is better than the other. This argument used in this experiment is only valid when degrading is allowed, that is, when parameter `intensify-only` is set to `False`, which is the general case.

Results from the experiment are depicted on table 5.13. We show that there is no real difference between the `CONSTANT` and the `LINEAR` scheduling of temperature.

Results obtained when benchmarking the `intensify-only` parameter are related to those. Indeed, we determined that allowing to degrade had few to no influence on tree quality over the course of the executions. For that reason, allowing to degrade has almost no influence, whether it is with the same probability during

Table 5.13: Influence of the `temperature schedule` parameter

Edge selection:	`RANDOM`		`MOST EXPENSIVE PATH`		`MOST EXPENSIVE EDGE`		`Greedy`
Schedule:	`CONSTANT`	`LINEAR`	`CONSTANT`	`LINEAR`	`CONSTANT`	`LINEAR`	
medium	1.0	1.002	0.995	0.996	1.0	1.0	*1.153*
	1.003	1.003	0.997	0.995	1.002	1.0	*1.107*
	0.995	0.997	0.995	0.991	1.002	1.0	*1.063*
stable	1.003	1.002	1.004	1.005	1.001	1.0	*1.092*
	1.006	1.006	1.008	1.01	1.0	1.0	*1.088*
	1.008	1.008	1.015	1.011	0.999	1.0	*1.099*
unstable	1.01	1.01	1.0	0.997	0.997	1.0	*1.094*
	1.014	1.013	1.007	1.002	1.002	1.0	*1.081*
	1.011	1.012	1.002	0.998	1.001	1.0	*1.069*
Geometric mean	**1.005**	**1.006**	**1.003**	**1.001**	**1.0**	**1.0**	***1.094***

the whole search, corresponding to the `CONSTANT` case, or with a decreasing probability, which corresponds to the `LINEAR` case. The impact of the `temperature schedule` parameter is therefore limited.

p `maximum selected paths`

The p parameter is used for selecting a path to remove, in the first part of the improvement step. It is only with the `MOST EXPENSIVE PATH` selection heuristic, as it works on the `PathQueue`. Because none of the configuration of the following experiment corresponds to our reference configuration, we do not set a particular column to be the reference, and let the algorithm select the best performing one instead.

To select a path to remove, the p most expensive paths are selected from the `PathQueue`, and one is chosen randomly. It therefore constitutes an additional source of diversification.

Table 5.14: Influence of the `maximum selected paths` parameter

Edge selection:	Edge selection: `MOST EXPENSIVE PATH`							`Greedy`
`maximum selected paths:`	**1**	**5**	**10**	**20**	**50**	**100**	**200**	
medium	1.001	1.0	1.001	1.005	1.018	1.031	1.029	*1.16*
	1.003	1.0	1.0	1.006	1.021	1.036	1.04	*1.116*
	1.001	1.0	1.002	1.006	1.019	1.033	1.032	*1.077*
stable	1.005	1.001	1.0	1.0	1.01	1.015	1.03	*1.097*
	1.01	1.001	1.0	1.003	1.01	1.019	1.027	*1.09*
	1.013	1.001	1.0	1.002	1.007	1.025	1.023	*1.098*
unstable	1.0	1.002	1.005	1.012	1.022	1.035	1.036	*1.099*
	1.0	1.001	1.001	1.004	1.015	1.021	1.027	*1.077*
	1.0	1.002	1.006	1.01	1.017	1.032	1.026	*1.071*
Geometric mean	**1.004**	**1.001**	**1.002**	**1.005**	**1.016**	**1.027**	**1.03**	***1.098***

Table 5.14 holds the results obtained in this experiment. We assume that a low value of the parameter tends to perform best, although we did not perform an extended statistical analysis.

That means that selecting the paths to remove among few of the most costly ones in an improvement round works better.

Indeed, using a high value for this parameter reduces the impact of using a heuristic for selecting which paths to remove. Heuristic which, by itself, is already computationally expensive, as some data structures such as the `PathQueue` must be maintained. In fact, if the value of the parameter is high enough, the heuristic amounts to randomly selecting a path among all of them, and it would just be wasting time to order paths according to their costs.

`Search strategy`

An improvement step is made of two parts. First, a path is selected and removed from the tree. Then, a path is found to reconnect the two created connected components. Parameter `search strategy` changes the behaviour of the reconnection process.

We defined two ways of exploring the space of reconnection paths. When parameter `search strategy` is set to `FIRST IMPROVEMENT`, the reconnection process stops as soon as a reconnection path whose weight is lower than the removed one is found. Choosing `FIRST IMPROVEMENT` therefore brings diversification. On the other hand, when value `BEST IMPROVEMENT` is selected for the parameter, the reconnection process takes into account all the nodes from the upstream connected component, that is, the component holding the root of the tree. This corresponds to intensification.

Table 5.15: Influence of the `search strategy` parameter

Edge selection:	`RANDOM`		`MOST EXPENSIVE PATH`		`MOST EXPENSIVE EDGE`		`Greedy`
Strategy:	`FIRST`	`BEST`	`FIRST`	`BEST`	`FIRST`	`BEST`	
medium	1.006	1.003	0.995	0.995	1.001	1.0	*1.155*
	1.002	1.0	0.996	0.996	1.0	1.0	*1.107*
	0.996	0.991	0.989	0.987	1.001	1.0	*1.062*
stable	1.002	1.001	1.008	0.998	0.999	1.0	*1.092*
	1.006	1.007	1.013	1.009	1.0	1.0	*1.089*
	1.014	1.007	1.014	1.01	1.0	1.0	*1.1*
unstable	1.015	1.011	1.001	0.996	1.002	1.0	*1.095*
	1.014	1.012	1.005	1.003	1.002	1.0	*1.081*
	1.015	1.011	1.006	1.001	1.005	1.0	*1.071*
Geometric mean	**1.008**	**1.005**	**1.003**	**0.999**	**1.001**	**1.0**	***1.094***

Seeing the results in table 5.15, we cannot affirm that one strategy is better than the other. Although we do not have any statistical evidence, it seems more interesting to use `BEST IMPROVEMENT`. The reason would be based on the thought that was given about parameter `intensify-only`, in which we concluded that intensifying systematically is no worse than degrading on some occasions. It is therefore meaningful to try to find the best possible solutions for each of the subproblems that compose the improvement step procedure, here, the reconnection procedure. By using `BEST IMPROVEMENT`, we ensure that the reconnection path that is found is the best one that could be found, while `FIRST IMPROVEMENT` further introduces diversification.

`Improve search space`

Just as for the `search strategy` parameter, the `improve search space` parameter affects the behaviour of the second part of the improve procedure, namely, the reconnection of the two created connected components.

This experiment aims at measuring the impact of narrowing the search space when reconnecting the two created components. The `improve search space` parameter acts as an upper bound for the number of nodes the reconnection process is allowed to consider while trying to find a better reconnection.

When reconnecting, we explore a subset of the nodes of the upstream created component, that is, the component that contains the root of tree. For each node of this subset, we consider all the nodes of the downstream component the upstream should connect to. The size of this subset of nodes is defined as the minimum between the size of the upstream components and the value given to the `improve search space` parameter. We can also chose to systematically include all nodes from the upstream component in the subset. This is the case when value is set to `ALL`.

Just as `search strategy`, `improve search space` is also a source of diversification.

Table 5.16: Influence of the `improve search space` parameter

	Edge selection: `RANDOM`						`Greedy`
Search space:	**1**	**5**	**10**	**25**	**50**	`ALL`	
medium	1.011	1.0	0.998	0.997	1.0	1.001	*1.152*
	1.018	1.002	0.998	0.999	0.999	1.003	*1.106*
	1.006	0.996	0.994	0.991	0.993	0.997	*1.063*
stable	1.011	1.005	1.003	0.999	0.999	1.0	*1.093*
	1.018	1.01	1.009	1.005	1.005	1.004	*1.089*
	1.01	1.009	1.006	1.005	1.005	1.007	*1.099*
unstable	1.015	1.009	1.008	1.005	1.007	1.011	*1.094*
	1.022	1.011	1.011	1.009	1.011	1.013	*1.082*
	1.018	1.008	1.006	1.007	1.004	1.008	*1.07*
Geometric mean	**1.014**	**1.006**	**1.004**	**1.002**	**1.003**	**1.005**	***1.094***

We would expect that reducing the `improve search space` prevents the algorithm from finding improving reconnection paths, because just as for preceding arguments, it makes sense to intensify as much as possible, i.e. setting a high value for this parameter, such as `ALL`. However, results depicted on tables 5.16 to 5.18 do not let us validate this assumption. We see that a low value does not clearly worsen the solutions.

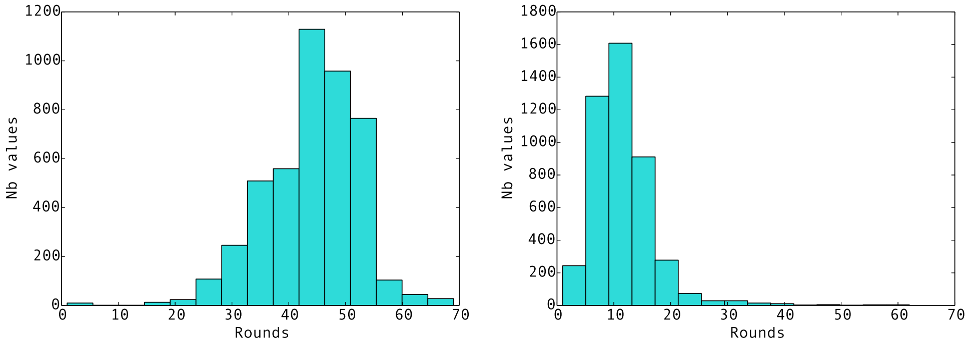
Table 5.17: Influence of the `improve search space` parameter

	Edge selection: `MOST EXPENSIVE PATH`						`Greedy`
Search space:	**1**	**5**	**10**	**25**	**50**	`ALL`	
medium	1.004	0.996	0.992	0.99	0.99	0.992	*1.152*
	1.009	0.997	0.992	0.991	0.99	0.994	*1.106*
	0.998	0.99	0.987	0.985	0.984	0.993	*1.063*
stable	1.006	0.998	0.997	0.996	0.997	1.002	*1.093*
	1.009	1.002	1.001	0.999	0.999	1.01	*1.089*
	1.008	1.002	0.999	0.999	1.0	1.016	*1.099*
unstable	1.006	1.002	0.997	0.996	0.995	0.994	*1.094*
	1.015	1.008	1.006	1.005	1.004	1.002	*1.082*
	1.017	1.004	1.004	0.999	1.0	0.999	*1.07*
Geometric mean	**1.008**	**1.0**	**0.997**	**0.996**	**0.995**	**1.0**	***1.094***

Table 5.18: Influence of the `improve search space` parameter

	Edge selection: `MOST EXPENSIVE EDGE`						`Greedy`
Search space:	**1**	**5**	**10**	**25**	**50**	`ALL`	
medium	1.004	0.996	0.995	0.994	0.997	1.0	*1.152*
	1.005	0.997	0.996	0.996	0.997	1.0	*1.106*
	0.998	0.991	0.991	0.992	1.0	1.0	*1.063*
stable	1.0	0.996	0.995	0.993	0.993	1.0	*1.093*
	1.009	1.0	1.001	0.998	0.998	1.0	*1.089*
	1.007	1.0	0.999	0.998	0.998	1.0	*1.099*
unstable	1.004	0.995	0.992	0.991	0.993	1.0	*1.094*
	1.011	1.003	1.001	0.997	0.998	1.0	*1.082*
	1.007	1.001	0.999	0.998	0.998	1.0	*1.07*
Geometric mean	**1.005**	**0.998**	**0.996**	**0.995**	**0.997**	**1.0**	***1.094***

This can be explained by the fact that limiting the explored search space allows the search to perform more rounds of improvement. Figure 5.8 depicts the number of rounds performed when the parameter is set to 1 and to `ALL`. We can see that the algorithm performs four to five times more rounds in the fig. 5.8a case than in the fig. 5.8b case. It is therefore able to try and improve more paths.



(a) `improve search space` = 1 **(b)** `improve search space` = `ALL`

Figure 5.8: Rounds of local search for different `improve search space` values

5.2.4 Effects of time allocated for improvement

In this experiment, we wanted to test the effects of allocating more time to improvement steps, while keeping constant the period at which those improvement steps are done. We aim at testing if allocating more time for improving yields better performing search algorithms, or if the benefits of improving fade as more time is given.

The following tests are thus "unfair" with respect to one another. Indeed, we purposely set a different `maximum improve time/tick` ratio to each of the configurations, unlike for previous tests, where this ratio was constant and maintained at 25 ms of time per `tick`. In the following tests, we increase this ratio up to 200 ms of improve time per `tick`.

Table 5.19: Influence of the `maximum improve time` parameter

Edge selection:	`RANDOM`			`MOST EXPENSIVE PATH`			`MOST EXPENSIVE EDGE`			`Greedy`
`ip/it:`	**`1/25`**	**`1/100`**	**`1/200`**	**`1/25`**	**`1/100`**	**`1/200`**	**`1/25`**	**`1/100`**	**`1/200`**	
medium	1.001	0.985	0.983	0.995	0.984	0.982	1.0	0.984	0.983	*1.155*
	1.001	0.98	0.978	0.994	0.978	0.977	1.0	0.98	0.977	*1.108*
	0.993	0.974	0.972	0.986	0.972	0.971	1.0	0.974	0.971	*1.063*
stable	1.002	0.992	0.992	0.999	0.99	0.989	1.0	0.991	0.99	*1.094*
	1.006	0.996	0.994	1.008	0.993	0.992	1.0	0.995	0.994	*1.09*
	1.01	0.997	0.997	1.009	0.996	0.995	1.0	0.997	0.996	*1.101*
unstable	1.012	0.98	0.975	0.998	0.975	0.973	1.0	0.978	0.974	*1.097*
	1.012	0.983	0.979	1.001	0.978	0.976	1.0	0.983	0.978	*1.082*
	1.007	0.98	0.977	0.996	0.976	0.974	1.0	0.98	0.976	*1.069*
Geometric mean	**1.005**	**0.985**	**0.983**	**0.999**	**0.982**	**0.981**	**1.0**	**0.985**	**0.982**	***1.095***

The results shown in table 5.19 are quite obvious: allocating more time to improvement steps (`maximum improve time` parameter) for a specific `improve period` systematically yields better solutions.

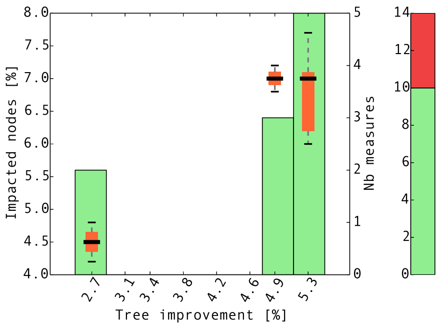
Another conclusion that can be drawn from this experiment is that the improving method needs a very small time to improve the tree by about ten percents, compared to the non-optimised `Greedy` configuration of the algorithm. By giving four or eight times more time to improve, the solution can be further improved by up to a few percents. We might argue that it is not needed to spend too much time on the improvement steps as a good improvement is quickly reached.

5.2.5 Impact of improvement steps on the network

In a real environment, improvement steps performed on the tree should not change the tree too much to avoid disturbing the network, especially if such improvement steps are performed on a regular basis. This requirement on the algorithm is important for it to be admissible in a real network.

For that reason, we want to measure the number of switches impacted by improvement steps, that is, which switches in the topology must be updated after such an improvement step is performed. We evaluate this impact for different scheduling periods and different amounts of allocated time.

We give a simple example of the representation of those measures at fig. 5.9.



On this simple example, the following improve and impacts are displayed:

Improve	Impact			
0	0	0	0	0
2.5	4.2			
2.8	4.8			
5	6.8	7	7.2	
5.2	6	7.1	7.7	
5.5	7.1	6.2		

The right plot represents the proportion of improvement steps that improved the tree. The red part represents the four unsuccessful improvement steps, and the green part, the others.

The bars on the left plot represents the number of measures per histogram bucket. In the example, there are more improvements around 5% than around 2.7%. The right axis of the left plot represents the number of measures. The sum of the bars is the same as the green part of the right plot.
The boxplots represent the distribution of the values in for each bucket and link the value of the improvements to the impact they have on the tree. The value of the impacts are given on the left vertical axis.

Figure 5.9: Simple example of the representation of the impact of improving

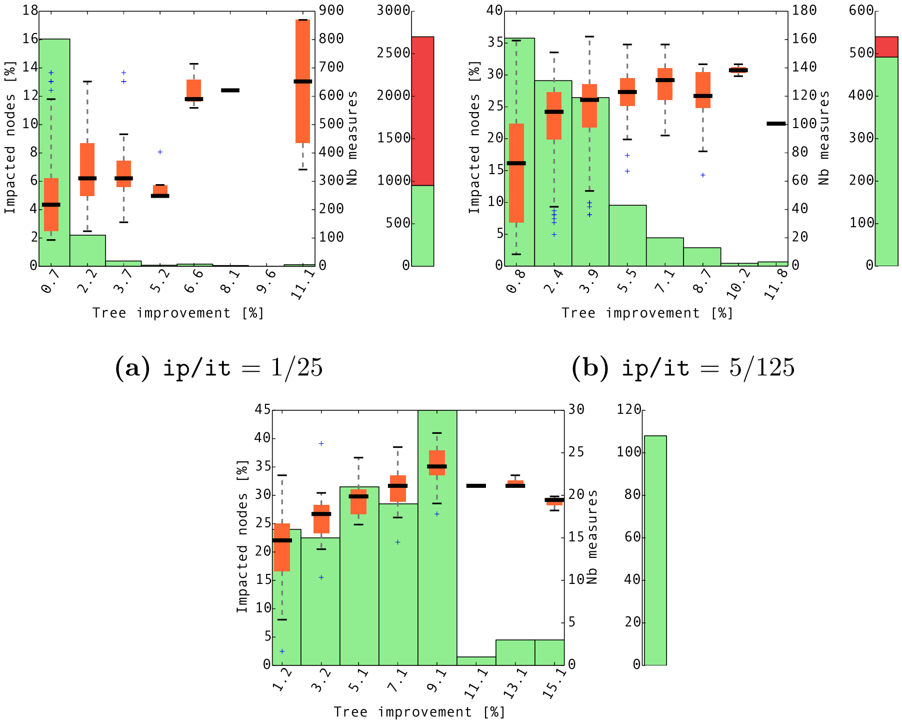
The ideal result would be to see big values of improvement having a small impact on the tree.

We collected values for three topologies, with several values of `improve period` and `maximum improve time`. The results are depicted on figs. 5.10 to 5.12.

For each topology, we can see that when the `improve period` is one (on the leftmost plots), a lot of improvement steps are wasted because they do not improve the tree. Because trying to improve takes time, it is better to schedule them when they are more likely to improve the tree.

By looking at the distribution of the measures, an observation we can make is that trying to improve less often, and thus giving more time, yields greater improvements. As a matter of fact, the distribution of the improvements along the horizontal axis shifts to the right when more time is given, especially on fig. 5.10 and fig. 5.12. The graphs show an evolution and should not be compared with each other without paying attention to the scales.

The left vertical axis shows the proportion of switches in the network that need to be updated after an improvement step. Depending on the used topology and on the `improve period` and `maximum improve time`, this proportion can get very high. This does not fit well in a real environment as we want to avoid disturbing



(a) ip/it = 1/25

(b) ip/it = 5/125

(c) ip/it = 25/625

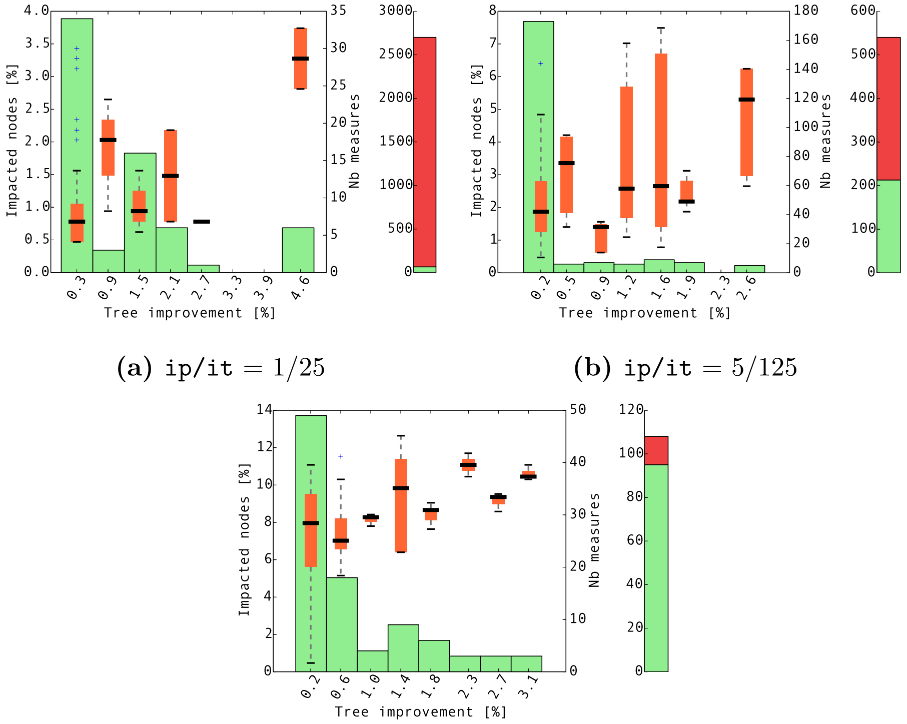
Figure 5.10: Impact of the improvement steps on the `Tiscali` topology

network switches too much upon an improve. Furthermore, if the improvement is small and the impact is big, it is not interesting to radically change the tree too often in order to gain an improvement of a fraction of percent to a few percents.

A way to limit this inconvenience would be to forbid improves that disturb the network too much, or to allow changes only when the improvement is significant enough. As an example, we might discard improvements whose effects on nodes is beyond a certain threshold, say, when they affect 20% of the topology. The second more complex way of discarding improvements might be to compare the level of improvement to the impact such an improvement has on the topology, say when the impact is at most twice the improvement. For instance, an improve of 5% might be allowed to update at most 10% of the network switches.

5.3 Conclusion of the experiments

In the previous sections, we evaluated the benefits brought by the `SDN` approach, which enables us to write tree building algorithms that know the whole topology. We specifically compared a `Greedy` algorithm, which does not schedule improvement steps yet, with the `PIM-SSM` approach, the current state of the art approach



(a) `ip/it` = 1/25

(b) `ip/it` = 5/125

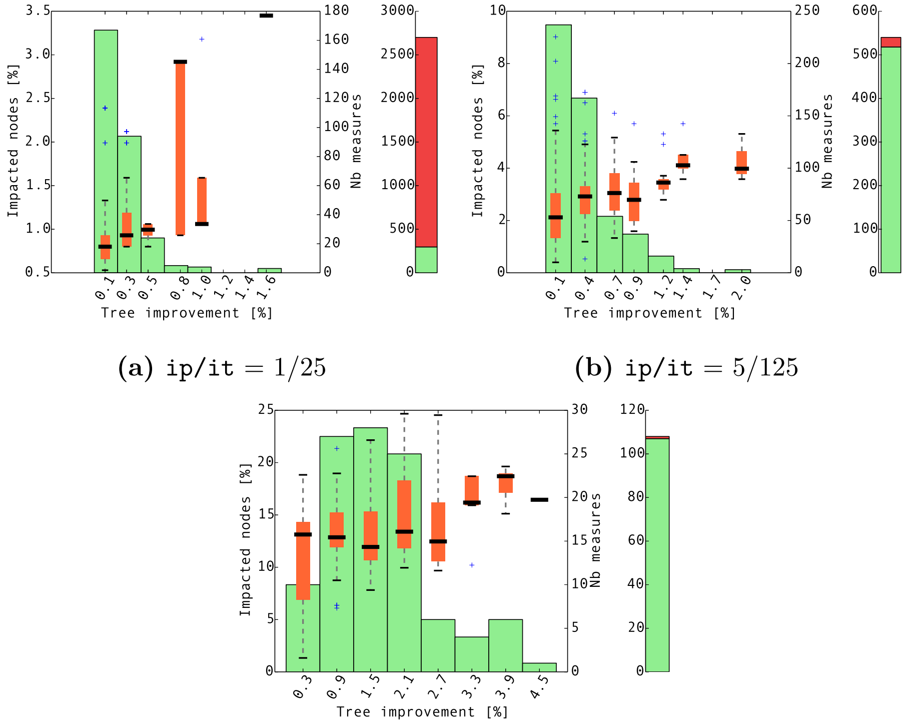
(c) `ip/it` = 25/625

Figure 5.11: Impact of the improvement steps on the `ANON1` topology

in classical networks. The improvement in tree cost, measured by the `averaged integral` metric in the `comparison table` representation, is of around 21% when measured on different scenarios running on the `Tiscali` topology.

In a second step, we measured the time required to handle multicast group events, such as `join` and `leave` events. The validation we made is that the time required to handle such events is linear with respect to the number of edges in the tree on which this event is applied. This result is important because we build our optimised method on top of the `Greedy` one. We thus have a basis that meets the network constraints expressed in terms of delay, which we aim at minimising.

In a third step, we considered scheduling improvement steps throughout the generation of the tree. We elected a specific algorithm configuration to further compare the benefits of improving on top of the `Greedy` approach used for handling network events. This specific algorithm configuration was optimised with respect to the different parameters. The impact of values for each parameter was measured in order to isolate this configuration, and justify our choice of value. This parameter optimisation process was done in section 5.2.3. It is based on one topology. We



(a) `ip/it` = 1/25

(b) `ip/it` = 5/125

(c) `ip/it` = 25/625

Figure 5.12: Impact of the improvement steps on the `KDL` topology

expect this best-performing configuration of the algorithm to slightly change with respect to the topology on which it is used. We therefore advise to find one such configuration per topology.

For each of the tests made on each parameter, we compared our three edge selection heuristics: the `RANDOM` selection, the `MOST EXPENSIVE PATH` selection and the `MOST EXPENSIVE EDGE` selection heuristics. Indeed, the impact of each parameter should be measured on a larger set of conditions. However, throughout our tests, no selection heuristic tended to perform better than the other.

A global conclusion that can be drawn with respect to the choice of arguments for the algorithm is that usually, values which favour diversification in spite of intensification tend to perform worse. As such, we limited the influence of such sources of diversification in our optimised configuration. Sources of diversification were identified in section 3.4.1.

The results of the testing of some parameters were not significant enough for choosing a specific value in spite of another. In particular, we determined that the `improve search space`, `search strategy`, `temperature schedule` and the `intensify-only` parameters had almost no influence on the tree cost over the course of an execution. We could not extract any real evidence of one value working better than another for such parameters. Values for such parameters were then chosen arbitrarily.

Regarding the `tabu time-to-live` parameter, we came to the conclusion that a `tabu list` is needed for two of the heuristics. Furthermore, we noticed that a `ttl` value of zero, which corresponds to the case where no `tabu list` is used, hinders the performance of heuristics which deterministically choose edges to remove, namely, the `MOST EXPENSIVE PATH` and `MOST EXPENSIVE EDGE` selection heuristics. The `RANDOM` edge selection approach does not suffer from setting a low value of `ttl`. Indeed, due to its non-deterministic nature, a `RANDOM` approach has no benefits from blacklisting its chosen edges in a `tabu list` as this choice is already made at random. We therefore defined a value of 50 for the `tabu time-to-live` parameter.

We considered the `maximum improve time` and `improve period` parameters together, as they are closely tied by the concept of fairness between different configurations. We consider a configuration to be fair with respect to another if the total time allocated for improvement steps over the course of the execution is the same. We defined a constant `maximum improve time/tick` ratio of 25 ms. In those tests, we iteratively increased the period at which improvement steps were scheduled, and validated several expected behaviours.

A first expected behaviour is that increasing the `improve period` increases the `averaged integral`. This is expected as this metric computes the integral of the tree after each `tick` interval. Improving less often, that is, after more `ticks`, thus increases this integral significantly. In order to reduce tree cost over the course of an execution, is makes sense to improve the tree as often as possible. This conclusion is valid for each of the selection heuristics.

A second expected behaviour is related to the number of rounds that can be performed for a given amount of time allocated to each improvement step. By plotting the amount of rounds of local search performed for each selection heuristic, we noticed that while the `comparison table` results from the different selection heuristics are pretty much alike, the different selection heuristics performed a significantly different number of rounds of local search per improvement step. In particular, we noticed that the computationally more expensive `MOST EXPENSIVE PATH` selection heuristic, which is based on the `PathQueue` data structure, performs nearly as good as the other heuristics, while performing a lower number of rounds of local search. The conclusion is that `MOST EXPENSIVE PATH` better guides the local search, but does it at a greater cost.

A third note must be made on the `maximum improve time/tick` ratio that is used. If such a ratio is too low, the benefits of improving more often but with a smaller allocated time might collapse. In such configurations, it might be more interesting to improve less often but allocating a comparatively bigger amount of time.

One of the argument bringing diversification is the `maximum selected paths` argument. It defines the number of most expensive paths among which to select one for removal, at each round of local search. This argument is therefore only used with the `MOST EXPENSIVE PATH` selection heuristic, and is of no use to other heuristics. The conclusion was here that the diversification brought by this argument does not improve the search. A value of 1 for this parameter seems to yield the best performing algorithms.

A last experiment was performed in order to quantify the benefits of allocating more time at each improvement step, the `maximum improve time` argument, for a given `improve period` value. The algorithm does not need much time to optimise the tree by a fair amount, and that giving more time only further improves by a few percents. We therefore decided to keep the ratio 25 ms per `tick` and a period of improvement step of 1 `tick`.

We measured the impact that improvement steps have on the whole topology, that is, the number of nodes affected by the change commanded by each improvement step. We plotted the level of improvement in relation to the impact it has on the whole topology. The experiments were carried for different values of `maximum improve time` and `improve period`. We draw several conclusions: many of the improvement steps have a great impact on the topology while only providing a small improvement. Also, when improving often, only a small proportion of the improvement steps succeed and only yield small improvements. It seems more reasonable to improve less often in order to find meaningful improvements. In order to limit the impact on the tree, we could further decide to filter improvement steps which do not sufficiently improve the tree.

$$
\begin{aligned}
\texttt{maximum improve time} &= 25\,\mathrm{ms},\\
\texttt{improve period} &= 1,\\
\texttt{improve search space} &= \infty\ (\texttt{ALL}),\\
\texttt{intensify-only} &= \texttt{False},\\
\texttt{search strategy} &= \texttt{BEST IMPROVEMENT},\\
\texttt{edge selection heuristic} &= \texttt{MOST EXPENSIVE EDGE},\\
\texttt{tabu time-to-live} &= 50,\\
\texttt{temperature schedule} &= \texttt{LINEAR}
\end{aligned}
$$

Figure 5.13: Optimised configuration

A set of optimised parameters have been used throughout the evaluation of our solution, since fig. 5.4. A recapitulation of the default values for parameters of our optimised configuration can be found in fig. 5.13. As the heuristic that has been selected is `MOST EXPENSIVE EDGE`, the parameter `maximum selected paths` is not applicable.

The results of all the experiments as well as the testing setups can be found on the a public repository [31].

Chapter 6

Conclusion

In this thesis, we dealt with the problem of computing multicast distribution trees under the assumption that we are in a software-defined network (`SDN`) environment. We used a method based on local search for building distribution trees in an online environment, where `join` and `leave` events happen at arbitrary moments. We implemented our method and tested it on realistic topologies, in order to measure its applicability in a real network.

We first summarise the content of this thesis and then highlight the main aspects.

Second, we discuss the limitations of our method and the issues that need to be overcome.

Third, we give leads on future work on the subject and other areas of the problem that can be studied.

We end the conclusion by bringing forward the issues that we had to face during the whole thesis and the lessons we learned.

6.1 Summary

Multicast is largely used for efficiently distributing data in today's networks. Software-defined networking (`SDN`) is a new and efficient approach to handle networks. The centralised approach brought by `SDN` gives a better knowledge of the network. The whole topology is known by a central entity. New more efficient methods for achieving multicast can then be explored based on this complete knowledge. In our thesis, we developed a way of building multicast distribution trees that benefits from the advantages brought by `SDN`, which are not present in classical networks.

Recently, a team of Taiwanese researchers released a preprint [40] which tackles the same multicast problem, but in a different manner.

Several researchers tackled either the problem of multicast in SDN or the issue of building optimised multicast trees with a particular knowledge of the topology. In any case, several limitations prevented us from exploiting their results.

When SDN was considered in the papers, there was often no mention of optimisation. Their research was about actual integration of solutions in a network, such as how to efficiently forward the packets or how to handle failures in an efficient way.

On the other hand, researchers aiming at optimising multicast or Steiner trees generally rely on some hypothesis that are not met in realistic networks, such as the triangular inequality, making their contribution unlikely to be applicable.

We developed a method for maintaining multicast distribution trees with online multicast group change events, such as `join` and `leave` events. To do so, we exploited the complete knowledge of the graph brought by the SDN approach.

The method consists in three main steps: the addition and removal of clients, and the possibility to improve the tree at regular intervals. We handle client additions and removals greedily, while avoiding to disturb the tree too much at such occasions. To try to improve the tree, we use local search. This choice is motivated by the fact that the search space is too big to be explored exhaustively. The process of searching for an improvement can be scheduled at any time while the tree is maintained and the events processed.

Without trying to improve, SDN enables us to maintain trees that are more efficient than the ones computed with current protocols.

To be able to test its applicability in a real network and its performance, we implemented our method with the `Python` language and libraries such as `NetworkX` or `pickle`. We extended available data structures to represent networks and multicast trees. In addition to those, our method imports and exports shortest paths of a network and maintains specific queues aimed at speeding up the improvement steps.

As our goal is to propose a more efficient method to build multicast trees, we want to test it against trees computed using classical solutions, such as PIM-SSM.

As we lacked real information on events occurring in multicast groups, we generated lists of events using a probabilistic model. Several metrics and variables were used in our experiments. The first one is aimed at measuring the quality of the tree over the course of the execution. The second is a measure of time to assess the possibility to use our method in a real environment. The third one measures the impact of improvement steps on the trees. It is also related to the applicability of our method in a real network.

The conclusions that were drawn from the experiments are that the algorithm gives better solutions than the current state of the art. This is due to the knowledge brought by the SDN approach, as well as to the improvement steps carried out over the course of the execution, which can further improve the solutions by a fair amount. Our algorithm performs fast enough to be used in real networks, however, the cost of improving in terms of modifications to the network can get high.

Finally, we discussed how to integrate our method in an actual SDN controller. We identified several requirements to meet in order to enable multicast in an OpenFlow network. We discussed the versions of OpenFlow and the features they offered.

We discussed about what should be added to our current implementation for it to be integrable in a real network. We also explained how improvement steps should be handled such as to provide some level of quality of service on the processing of network events.

6.2 Limitations

Our solution computes better distribution trees than the PIM-SSM method. Nonetheless, the solution we provide has several limitations. Some are related to the method we developed, while others come from the nature of the problem.

6.2.1 Limitations of our method

It was shown that applying improvements on the tree might imply that a large proportion of network devices have to be updated. Guards should be set in order to avoid disturbing the network too much for non-significant improvements.

Another possible improvement of our approach would be to determine when improving the tree is needed and should be scheduled. We could do so by keeping track of events occurring in the network or by keeping track of the evolution of the tree weights. For instance, an improve might be scheduled once the cost of the tree reaches a threshold.

A second limitation inherent to our method is the need to pre-process and keep in memory the shortest paths between each two nodes of the graph. Although it reduces the overall complexity of the method, it limits the scalability of the approach as space complexity is cubic with respect to the number of nodes in the graph.

6.2.2 Limitations inherent to the problem

Due to the NP-complete nature of the minimum Steiner tree problem, it is impossible to compute optimal solutions without exploring the whole search space, even with the knowledge brought by the `SDN` approach. As we work with large topologies containing several hundreds of nodes, we cannot by any mean compute the optimal tree. For that reason it is not possible to assess the quality of the tree we build compared to an optimal Steiner tree.

The tree could be build in a more efficient way if all the events are known in advance. However, the online nature of the problem prevents us from doing so.

6.3 Future work

We begin with a discussion about how our solution can be implemented in the `OpenFlow` `SDN` environment. We list the features that `OpenFlow` must support.

We then describe how our local search approach should be integrated in a controller: we suggest ways to process network events and schedule improvement steps during the course of the multicast service.

We then list some features that have not been covered by our method, but from which it could benefit. For each of those concerns, we elaborate some leads for solving them.
The first concern we want to address is about failures. The second one is more related to the applicability of the solution so as to manage several multicast groups on the same network.

6.3.1 Integration in an `OpenFlow` network

For our method to be exploitable, `OpenFlow` should support several features that are mandatory to achieve multicast. We explain those features of `OpenFlow` here below and give insights on what to use to implement our solution in a real environment.

Required features of `OpenFlow`

In order to support multicast, the switches and controller must support the following features:

Packet duplication In order to support efficient multicast, as described in section 1.1.3, packets being routed in the network might need to be duplicated and forwarded over several ports. In order to handle duplication, `OpenFlow 1.0` is necessary, which enables the use of multiple actions of the same type, via the use of `Action Lists`.

An Action List contains several actions that must be applied when forwarding certain packets. In such a list, more than one "output action" can be specified. This feature is available since OpenFlow version 1.0.

Group membership management The management of multicast groups, that is, the association between a sender and a set of subscribers. A set of nodes from the network might choose to join or leave such groups.

Managing those groups is achieved using different protocols in IPv4 and IPv6.

Using IPv4: the group management protocol is the Internet Group Management Protocol [23] (IGMP).

Using IPv6: IPv6 mechanisms for supporting group memberships include Multicast Listener Discovery protocol [23, 41] (MLD), a sub-protocol of Internet Control Message Protocol version 6 [42] (ICMPv6).

In an OpenFlow network, those group management messages should be handled by the controller. Group management packets should then be matched and forwarded to the controller. This is achieved using *snooping*.

SNOOPING

IPv6 support was introduced in OpenFlow specifications 1.2 and was further refined in specifications 1.3. Anything using a version of OpenFlow below 1.2 will not be compatible with IPv6.

In IPv4, rules must match on the IGMP protocol. IGMP runs as a separate protocol. It is thus easy to match on this kind of packets, by looking at the Protocol field of the IPv4 header. In practice, the snooping of such packets is achieved by defining the following entry in a flow table:

***RULE*: (proto == IGMP) ⟹ *ACTION*: (output: CONTROLLER)**

This entry is understood by all stable versions of the OpenFlow specifications.

In IPv6, rules must match on the MLD protocol, which is a sub-protocol of the ICMPv6 protocol. As opposed to IGMP, MLD does not run as a separate protocol. Matching on the corresponding Next Header field of the IPv6 header is not sufficient, as we would match on all kinds of ICMPv6 packets, and not on MLD packets in particular.

In order to match MLD packets, we need to match on the Next Header field of the IPv6 header. In version 1.2 of the OpenFlow specifications [17], published by The Open Networking Foundation [43] , the field OFPXMT_OFB_ICMPV6_TYPE supports matching on the ICMPv6 message type. Unfortunately, this field is not in the minimum required fields, meaning that we first need to test if flow tables actually support this field before trying to define rules for matching on it. When using IPv6, the controller should first verify that a node that can become a client is able to match on this field, in order to forward its group management traffic to the controller. `OpenFlow` defines mechanisms to perform such verification.

Feature	IPv4	IPv6
Duplication	OF1.0	
Group management	IGMP snooping OF1.0	MLD snooping OF1.2 (+ support of non-required field)

Figure 6.1: `OpenFlow` minimal requirements for multicast following IP version

Some controllers servicing multicast already exist. For example, CastFlow [24], which as of now does not support a specific group management protocol. It was extended in Groupflow [44]. CastFlow was described in section 2.2.2. The authors of the Groupflow project added `IGMP` version 3 support to the CastFlow controller, making it usable with today's IPv4 network devices.

Some controllers, such as Ryu [45], support `IGMP` and `MLD` natively. It could then be used as baseline controller for building a multicast-enabled controller.

6.3.2 Local search in a real network

In this section, we cover two aspects that must be dealt with when using our local search approach in a real network. Those are how to handle network events in an interactive fashion and how to schedule improvements over the course of the execution. Additional components must be included in the solution if we want to maintain constraints from the network.

Interactive usage of the algorithm

Our current program implements the algorithms and its requirements on the network. However, because we directed its implementation towards its usage in a testing environment, where network events are pre-computed before the program is actually run, there lacks routines for handling events in a real-time fashion. This could for example be implemented easily by first detecting multicast group change events and then process them from a single first-in first-out (FIFO) queue. They would be processed in order of their arrival, always on the same tree.

In order to maintain the tree competitive, improvements steps would then have to be scheduled in between those events.

Another way of handling the events is to maintain two FIFO queues, one for the join events and one for the leave events. Using two queues could allow to prioritise the join requests over the leave ones.

Scheduling improvement steps

As mentioned in section 3.4, improvements can be performed any time between network events. Meaning that we can choose when to perform them and for how long.

However, during an improvement step, the algorithm locks the multicast tree and does not allow `join` and `leave` events to be applied. Depending on the time allowed for improving, we need to tackle the execution of the improving steps differently.

Whatever the time taken by an improvement step, we want the `join` and `leave` events to be treated within an acceptable time right after their occurrence. There are two cases, when an improvement is set to take a short time, and when it takes more time.

Fast improvement steps : If a short time is allocated for improving the tree, events arriving while the improvement is being performed can be buffered and applied on the optimised tree when the improvement is over.

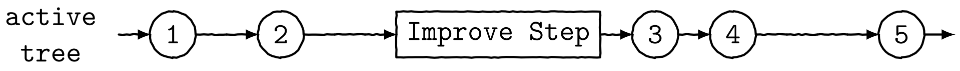


Figure 6.2: Sequential improve

As can be seen on fig. 6.2, the events are buffered until the current improvement step finishes. This could be the case of ③ and ④. Those events are then applied on the improved tree. This sequential processing scheme can suffice if improvement steps are short enough.

Longer improvement steps : When improvements are set to take a longer time, buffering the events while waiting for the optimisation to be over can lead to events being applied a long time after they appear on the network.

A solution for handling such situations is to perform the improvement steps on a separate copy of the active tree, while still handling events on the active tree. Once the improvement step is finished on the separate tree, events that arrived during the optimisation and that have been processed on the active tree must also be processed on the separate tree. Once both trees achieve the same service (all events have been processed), the active tree can be replaced by the improved one.

This process of *parallel improvement* is depicted on fig. 6.3. Events ① to ⑤ have to be applied to the tree. At step Ⓓ, the active tree is duplicated and an improvement step is started on the copy of the tree. In this case, the improvement step takes way more time than applying a few events.

PARALLEL IMPROVEMENT

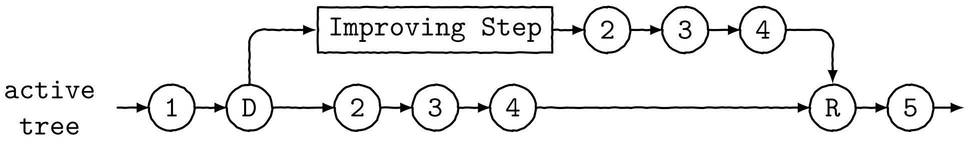


Figure 6.3: Parallel improve

Events ②, ③ and ④ occur while the improvement is being processed. They need to be both buffered and applied on the active tree such as to be processed directly. This low delay for handling multicast service event is a requirement that we previously defined. When the improvement is done, those buffered events must be applied to the optimised tree for it to achieve the same service as the active tree.

An event that would appear when the buffered events are being applied to the optimised tree should be treated the same way (buffered and applied to the active tree).

When the buffer is empty, the events have been applied to both trees, and thus they achieve the same service. The active tree is then replaced by the optimised one at step Ⓡ.

6.3.3 Failure-recovery

Failures are an important aspect that must be taken into account in every real set-up. Our solution does not react to failures.

LINK FAILURES
SWITCH FAILURES

Here, we distinguish several types of failures that might occur in the network: *link failures* and *switch failures.* The latter case of failure can be expressed in terms of the first one. Indeed, a failing switch is equivalent – to our search algorithm at least – to having all adjacent links of this switch failed.

We believe that those failures can be taken into account in our solution. Our algorithm is based on a pre-computation phase of shortest paths between each nodes in the network. This pre-computation phase is done on the assumption that no failures are occurring in the network. A way of taking the failing links into account would be to maintain a list of such failed links and recompute only the shortest paths including it. This correction can be done as soon as the failure is detected, or in a lazy manner.

Either way, the graph structure can be further expanded to take the corrected shortest paths into account.

Computing more than one shortest path between each nodes of the graph – typically *k* shortest paths – would provide us with alternative paths not using certain links of the tree. However, this simple solution would not be robust enough to cover failures of any of the links of the network.

Extending our method to be able to react to failures could use the same mechanism as the improving process. Instead of choosing an edge or a path to remove in an improvement step, the edge would be imposed as the failing link. Then, selecting a reconnection path would work the same way as it does already with the additional constraint that the reconnection path cannot contain any defective link.

6.3.4 Management of several multicast groups

Computing several distribution trees can be done by using our algorithm several times, once for each multicast group. The problem that might arise is that the load on the links might become too high. On specific network configurations, the algorithm might tend to use the same links for each multicast trees.

Overcoming that issue could be done by forbidding some of the multicast trees to used overloaded links in the network. Similarly to treating the failure, it would question the pre-computation of the shortest paths in the networks.

6.4 Retrospective

Throughout the development of our method and the writing of the thesis, we had to deal with several difficulties and learned some valuable lessons.

A difficulty we had to overcome and that we rarely encountered during our university education is that we did not rely on any existing implementation. Due to the nature of our problem, we could not build our implementation on top of an existing solution. Despite this issue, solving the problem described in this thesis made use of concepts introduced in lectures, such as optimisation methods and network concepts. Doing this thesis offered us the occasion to practically use concepts coming from the two paradigms.

While implementing, we realised more than ever the necessity to automate our actions as much as possible. In fact, it is inconceivable in large enough projects to need to change bits of codes to change the behaviour of the algorithm. For that matter, we ended up creating lots of parameters, configuration files and scripts enabling us to adapt, with the least possible modifications, the behaviour of the algorithm and the testing procedures.

The presented solution evolved tremendously throughout its lifetime. At first, we were hesitant regarding rewriting or modifying working functions and bits of code, and would rather adapt to the existing content rather then re-factoring the necessary bits to get easier and cleaner structure. However, we quickly had to realise that the evolving solution questioned a lot of what was already produced. Throughout the addition of functionalities, we re-factored more and more, and still, some part of the implementation would gain from being rewritten.

We were not used to the good practice to apply when it came to evaluate our solution. One example of this fact is that, at first, we did not consider the reproducibility of the experiments performed on the algorithm. Results from our first experiments were impossible to replicate. Moreover, the datasets on which experiments were run were not realistic. We therefore learned to keep track of all data on which our results are based, as well as how to generate realistic datasets.

As this thesis primarily a research work, we did not have precise specifications on what was to be done, unlike in most of the projects we did earlier, where the goal to achieve was well defined. We were not sure what would come out of the method we implemented and we did not know what results to expect. We had for example hoped that the `MOST EXPENSIVE PATH` heuristic along with the usage of a `PathQueue` would provide us with good performance. Seeing that it did not perform better compared to simpler heuristics was somehow a disappointment. However, we learned that not having a difference in performance between several approaches is already a result in itself. The same conclusion holds for the different parameters of our algorithm, some of which did not improve the results by the expected amount.

During the thesis, our way of computing the tree and of seeing the problem changed. Because of that evolution, we implemented many side-scripts and interfaces to other tools that were abandoned in the final version. Among those, we implemented several scripts using the `R` language [46] to format the results of the experiments. We designed tests that were not relevant enough. We also made use of algorithm configurators such as `irace` [47]. In the early versions of the implementation, our method did not include a `PathQueue`, and so, functions to identify the paths in the tree had to be implemented.

Regarding the global method, we wanted at first to try and improve the tree after each addition and removal of nodes. The method we used in the end is completely different as it consists of small improvements steps based on local search which can be scheduled at any time.

On a final note, the thesis was an additional occasion for us to work in pair on a single project. We believe that working together was a strong asset in the realisation of this work.

Bibliography

[1] Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM J. Discrete Math.*, 4(3):369–384, 1991.

[2] Vachaspathi P. Kompella, Joseph C. Pasquale, and George C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Trans. Netw.*, 1(3):286–292, June 1993.

[3] Eduardo Uchoa and Renato F. Werneck. Fast local search for the steiner problem in graphs. *J. Exp. Algorithmics*, 17:2.2:2.1–2.2:2.22, May 2012.

[4] Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. The power of recourse for online mst and tsp. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'12, pages 689–700, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Albert Gu, Anupam Gupta, and Amit Kumar. The power of deferral: Maintaining a constant-competitive steiner tree online. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 525–534, New York, NY, USA, 2013. ACM.

[6] Kok-Kiong Yap, Te-Yuan Huang, Ben Dodson, Monica S. Lam, and Nick McKeown. Towards software-friendly networks. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 49–54, New York, NY, USA, 2010. ACM.

[7] Olivier Bonaventure. LINGI2141: Computer networks : information transfer. `http://www.uclouvain.be/cours-2013-LINGI2141`, 2014.

[8] Olivier Bonaventure. LINGI2142: Computer networks: configuration and management. `http://www.uclouvain.be/cours-2013-LINGI2142`, 2014.

[9] An internet multicast system for the stock market. *ACM Trans. Comput. Syst.*, 19(3):384–412, August 2001.

[10] IETF. Protocol independent multicast. `http://datatracker.ietf.org/wg/pim/`, 2014.

[11] S. Bhattacharyya. An Overview of Source-Specific Multicast (SSM). RFC 3569 (Informational), July 2003.

[12] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[13] The Open Networking Foundation. Openflow specs 1.0. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`.

[14] The Open Networking Foundation. Openflow specs 1.4. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.pdf`, October 2013.

[15] Nicira Networks. Open vswitch: An open virtual switch,"http:// openvswitch.org, 2014.

[16] The Open Networking Foundation. Openflow specs 1.3. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`, June 2012.

[17] The Open Networking Foundation. Openflow specs 1.2. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf`, December 2011.

[18] Point-Topic. Analysis of iptv. `http://point-topic.com/free-analysis-technology/iptv/`, 2014.

[19] Amin Vahdat. Enter the andromeda zone - google cloud platform's latest networking stack. `http://googlecloudplatform.blogspot.be/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html`, 2014.

[20] Jim Duffy. Cisco reveals openflow sdn killer. `http://www.networkworld.com/news/2014/040214-cisco-openflow-280282.html`, 2014.

[21] F.K. Hwang, D.S. Richards, and P. Winter. *The Steiner Tree Problem.* Annals of Discrete Mathematics. Elsevier Science, 1992.

[22] Yves Deville. LINGI2261: Artificial intelligence: representation and reasoning. http://www.uclouvain.be/en-cours-2013-LINGI2261, 2014.

[23] H. Holbrook, B. Cain, and B. Haberman. Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast. RFC 4604 (Proposed Standard), August 2006.

[24] C.A.C. Marcondes, T.P.C. Santos, A.P. Godoy, C.C. Viel, and C.A.C. Teixeira. Castflow: Clean-slate multicast approach using in-advance path processing in programmable networks. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 000094–000101, July 2012.

[25] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '01, pages 346–, Washington, DC, USA, 2001. IEEE Computer Society.

[26] Lucas Bondan, Lucas Fernando Müller, and Maicon Kist. Multiflow: Multicast clean-slate with anticipated route calculation on openflow programmable networks. *Journal of Applied Computing Research*, 2(2):68–74, 2013.

[27] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[28] D. Kotani, K. Suzuki, and H. Shimonishi. A design and implementation of openflow controller handling ip multicast with fast tree switching. In *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on*, pages 60–67, July 2012.

[29] Ieee standard for local and metropolitan area networks– station and media access control connectivity discovery. *IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005)*, pages 1–204, Sept 2009.

[30] Trema controller team. Trema project website. http://trema.github.io/trema/.

[31] Kevin Jadin and Debroux Léonard. Public repository for implementation and experiment results. http://thesis.kjadin.com, 2014.

[32] Python Software Foundation. Python 2 documentation. https://docs.python.org/2/.

[33] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[34] Platoscave. Calculate distance between latitude longitude pairs with python. `http://www.platoscave.net/blog/2009/oct/5/calculate-distance-latitude-longitude-python/`, 2014.

[35] J.Y. Yen. Finding the k shortest loopless paths in a network. *management Science*, pages 712–716, 1971.

[36] Python Software Foundation. Python: Time access and conversions. `https://docs.python.org/2/library/time.html#time.clock`, 2014.

[37] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

[38] University of Washington. Rocketfuel: An isp topology mapping engine. `http://www.cs.washington.edu/research/networking/rocketfuel/`, may 2014.

[39] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[40] Liang-Hao Huang, Hui-Ju Hung, Chih-Chung Lin, and De-Nian Yang. Scalable steiner tree for multicast communications in software-defined networking. *CoRR*, abs/1404.3454, 2014.

[41] R. Vida and L. Costa. Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810 (Proposed Standard), June 2004. Updated by RFC 4604.

[42] R. Bonica, D. Gan, D. Tappan, and C. Pignataro. Extended ICMP to Support Multi-Part Messages. RFC 4884 (Proposed Standard), April 2007.

[43] The Open Networking Foundation. Website. `https://www.opennetworking.org/`.

[44] Alexander Craig. Groupflow. `http://alexcraig.github.io/GroupFlow/`.

[45] Ryu controller. `http://osrg.github.io/ryu`.

[46] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[47] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.