CCIT Report #417
March 2003

Precomputation Schemes for QoS Routing

Ariel Orda *Alexander Sprintson*
Department of Electrical Engineering
Technion—Israel Institute of Technology
Haifa 32000, Israel

Abstract

Precomputation-based methods have recently been proposed as an instrument to facilitate scalability, improve response time and reduce computation load on network elements. The key idea is to effectively reduce the time needed to handle an event by performing a certain amount of computations in *advance*, *i.e.*, prior to the event's arrival. Such computations are performed as background processes, thus enabling to promptly provide a solution upon a request, through a simple, fast procedure.

In this report, we investigate precomputation methods in the context of QoS routing. Precomputation is highly desirable for QoS routing schemes due to the high computation complexity of selecting QoS paths on the one hand, and the need to promptly provide a satisfactory path upon a request on the other hand. We consider two major settings of QoS routing. The first is the case where the QoS constraint is of the "bottleneck" type, *e.g.*, a bandwidth requirement, and network optimization is sought through hop minimization. The second is the more general setting of "additive" QoS constraints (*e.g.*, delay) and general link costs.

The report mainly focuses on the first setting. First, we show that, by exploiting the typical hierarchical structure of large-scale networks, one can achieve a substantial improvement in terms of computational complexity. Next, we consider networks with topology aggregation. We indicate that precomputation is a necessary element for any QoS routing scheme and establish a precomputation scheme appropriate for such settings. Finally, we consider the case of "additive" QoS constraints (*e.g.*, delay) and general link costs. As the routing problem becomes $\mathcal{NP}$-hard, we focus on ε-optimal approximations, and derive a precomputation scheme that offers a major improvement over the standard approach.

Keywords– QoS, Routing, Precomputaiton, Hierarchical networks, Topology aggregation.

1 Introduction

In recent years we have witnessed considerable accomplishments in the design, development and deployment of broadband communication networks. Network capabilities, in particular those of the Internet, grow at a remarkable rate. At the same time, a phenomenal growth in data traffic and a wide range of new requirements of emerging applications, call for new mechanisms for the control and management of communication networks. This poses some major challenges. Not only are many problems intrinsically difficult, but there are also additional constraints, such as limitations on the computational capabilities of network elements. In addition, any such control and management mechanism must scale well with network growth, and provide fast response to internal (*e.g.*, link failure) and external (*e.g.*, connection request) events.

Precomputation-based methods have recently been proposed as an instrument for facilitating scalability, improving response time and reducing the computational load on network elements. The key idea is to effectively reduce the time needed to handle an event, by performing a certain amount of computations *in advance*, *i.e.*, prior to the event's arrival. Such advance computations are performed as background processes, *i.e.*, when a network element is idle or underutilized, thus resulting in better utilization of the computational capabilities of network elements. In addition, when the rate of external events is high, a considerable reduction in overall computational load is achieved.

Precomputation is performed by means of a two-phase procedure, which we refer to as a *precomputation scheme*. The first phase is executed in advance and its purpose is to precompute solutions *a priori* for a wide set of possible event parameters. The computations performed at this phase are then summarized in a database for later usage. The second phase is activated when an event arrives and its purpose is to promptly provide an adequate solution for the event's parameters. The second phase either selects one of the solutions precomputed at the first phase, or, if necessary, performs a few additional computations. For instance, when handling connection requests with delay constraints, the first phase may precompute paths for a wide range of possible delay constraints, while the second phase just needs to select a suitable path from the precomputed database, *i.e.*, one that satisfies the particular delay constraints of the connection. The execution time of the second phase has an immediate impact on network performance, hence it is highly desirable to keep its computational complexity as low as possible. In the above example, the less time consumed in finding the proper path, the less time is consumed in establishing the new connection.

We conclude that precomputation is a highly desirable scheme and, at times, a necessary component for the efficient control and management of broadband networks. We proceed to discuss its major benefits in some more detail.

Enhancing scalability. As networks grow in size, appropriate control mechanisms must scale well with network growth. The two major strategies for achieving scalability are limiting the amount of link state information, and reducing the computational load of network elements. Precomputation methods constitute a useful tool for both strategies. Indeed, in many typical settings, where the rate of event arrivals is high, precomputation allows reduction of the overall computational load. Furthermore, as shall be discussed below, precomputation methods are necessary for handling *topology aggregation*, a major technique for obtaining scalability.

Improving fault tolerance. A failure of a network element (*e.g.*, link or node) must be handled properly, for example, by re-routing the existing connections to alternative routes. Failures are handled much faster if some computations are performed in advance. For instance, an alternative path can be precomputed for each possible link failure.

Improving performance in bursty conditions. Under bursty conditions, a new connection request might arrive before the handling of a previous request has been completed. Computations performed prior to the burst reduce the time needed for handling a request.

Improving load balancing. In a precomputation scheme, a number of potential requests are processed through the same procedure. This facilitates distribution of available resources among different requests

in an efficient manner. For example, consider a setting in which packets are sent along shortest (or almost-shortest) paths, determined by the source node. For such a setting, a precomputation scheme would identify a number of shortest and near-shortest paths in advance and supply different paths to different connection requests, effectively facilitating load balancing.

In fact, several existing network mechanisms employ some form of precomputation. As a straightforward example, consider standard IP routing, where each packet is forwarded by a router according to its *precomputed* routing table.

As shall be demonstrated, many of the algorithmic tools that are often proposed as building blocks for network control and management were not designed with precomputation in mind, and better results can be obtained when such a scheme is considered. The problem of how to efficiently precompute a set of solutions for a wide range of parameters effectively opens a new area of research. We note that the running time of the precomputation scheme is important due to the following reasons. First, the time available for precomputation is limited because the network element has other off-line tasks. A second limitation arises from the need to invoke the precomputation scheme upon changes in the link state, because such changes may invalidate the precomputed solutions.

In this report, we focus on the precomputation perspective of QoS routing. QoS routing is, undoubtedly, one of the major building blocks for supporting QoS, and hence a necessary component of future communication networks. Indeed, it has been the subject of several recent studies and proposals (see, *e.g.*, [6, 7, 9, 12, 17, 20, 22, 24, 25, 27, 28] and references therein).

QoS routing is, in general, a complex problem, due to several reasons. One complication is the need to deal with several QoS requirements, each potentially imposing some constraints on the path choice. Then, beyond the need to address the requirements of individual connections, QoS routing needs to consider also the global use of network resources. The above obstacles notwithstanding, QoS routing is facilitated in many practical settings by the following. First, while a connection may pose several QoS requirements, it turns out that these often translate mainly into a *bandwidth* requirement [1, 2]. Bandwidth, in turn, belongs to the class of "bottleneck" path requirements, which are much easier to handle than "additive" requirements, such as delay, loss or jitter [12, 17, 18]. As for global network optimization, often it turns out that much can be achieved by employing the simple criterion of *hop minimization* [1, 3]; indeed, a consequence of the need to reserve resources such as bandwidth on *each* link of the connection's path is that with fewer hops, one consumes fewer resources. As a result, hop-constrained path optimization has emerged as an important component of several recent proposals for IP-oriented QoS routing protocols [9]. Luckily, hop minimization turns out to be an optimization criterion that is relatively easy to handle.

The high complexity associated with QoS routing on the one hand, and the requirement of fast path selection on the other hand, make precomputation highly desirable for QoS routing schemes. Accordingly, this report mainly focuses on the problem of precomputing paths of maximal bandwidth for each possible hop-count value. This problem was initially investigated in [13], and was termed there as the *All-Hops Optimal Path* problem (AHOP). While a trivial solution to that problem is offered by the standard Bellman-Ford algorithm [8], in [13] an algorithm with a lower worst case bound is presented; yet, the improvement is achieved only in dense (highly connected) topologies, while communication networks usually have a sparse topology. In this study, we show that, by exploiting the *hierarchical structure*, typical of large-scale networks, a better solution in terms of computation complexity can be obtained.

Next, we turn to consider QoS routing in networks with topology aggregation, which improves the scalability of link state protocols by effectively limiting the amount of link state information stored at a node. With topology aggregation, subnetworks, or *routing domains*, do not reveal the details of their internal structure, but rather supply the aggregated representation to the outside world. The aggregated representation summarizes traversal characteristics of a routing domain. It may include, for example, the description of paths available across the domain. The aggregated representation is not computed for a specific event parameter, such as required path delay, but for a wide range thereof; therefore, as we shall see, precomputation techniques are an efficient tool for calculating aggregated representations. We indicate that precomputation is a necessary element for performing QoS routing in such settings and

establish an appropriate precomputation scheme.

Finally, we demonstrate the (wide) scope of problems that can benefit from precomptuation techniques by considering the harder case of *additive* QoS requirements and *general* (additive) path optimization criteria (*i.e.*, other than hop minimization). The respective problem becomes a variant of the *Restricted Shortest Path (RSP)* problem, which is known to be $\mathcal{NP}$-hard [11]. Some general approximation schemes that are ε-optimal have been proposed (see, *e.g.*, [26] and references therein). However, those schemes have not been designed with precomputation in mind, and, consequently, are not adequate when precomputation is sought. Accordingly, in the present study we establish an approximation scheme, which offers both efficient solutions as well as efficient performance, for precomputing "optimal" (minimum cost) paths for all possible values of an additive QoS requirement.

The rest of the report is organized as follows. First, in Section 2, we formulate the network model and formally state the considered problems. In Section 3, we consider the problem of hop minimization with bottleneck QoS constraints in hierarchical networks, and present and analyze our precomputation scheme. In Section 4, we extend our scheme for networks with topology aggregation. In Section 5, we consider additive QoS constraints and general (additive) path costs; we present and analyze our precomputation scheme, and demonstrate its advantages over standard alternatives. Finally, conclusions appear in Section 6.

2 Model and Problem Formulation

This section formulates the general model and main problems addressed in this report.

2.1 Network Model

We begin with a definition of a *general* communication network; definitions of some specific classes, namely *hierarchical* and *aggregated* networks, will be introduced in the following sections.

A *network* is represented by a directed graph $G(V, E)$, where V is the set of nodes and E is the set of links. We denote by N and M the number of network nodes and links, respectively, *i.e.*, $N = |V|$ and $M = |E|$. An (s, t)-*path* is a finite sequence of distinct nodes $\mathcal{P} = \{s = v_0, v_1, ..., t = v_h\}$, such that, for $0 \leq i \leq h - 1$, $(v_i, v_{i+1}) \in E$; $h = |\mathcal{P}|$ is then said to be the *number of hops* (*or hop count*) of $\mathcal{P}$. The subpath of $\mathcal{P}$ that extends from v_i to v_j is denoted by $\mathcal{P}_{(v_i,v_j)}$. Let $\mathcal{P}_1$ be a (u_1, u_2)-path and $\mathcal{P}_2$ be a (u_2, u_3)-path; then, $\mathcal{P}_1 \circ \mathcal{P}_2$ denotes the (u_1, u_3)-path formed by concatenation of $\mathcal{P}_1$ and $\mathcal{P}_2$. We denote by H be the maximum possible hop count of any path $\mathcal{P}$ in G that may be considered for routing purposes. Obviously, $H \leq N - 1$, and it is much smaller in many typical network topologies.

Each link $e \in E$ is assigned a positive *weight* w_e, whose significance depends on the type of considered QoS requirement. For example, when the QoS requirement is an upper bound on the end-to-end delay, the link weight is its delay; whereas when a bandwidth requirement is considered, the link weight w_e is reciprocal to its available bandwidth. Accordingly, the *path weight* $W(\mathcal{P})$ of a path $\mathcal{P}$ is defined differently for additive metrics, such as delay, than for bottleneck metrics, such as bandwidth. When link weights constitute an *additive* metric, the *weight* $W(\mathcal{P})$ of a path $\mathcal{P}$ is defined as the sum of weights of its links, *i.e.*, $W(\mathcal{P}) = \sum_{e \in \mathcal{P}} w_e$. When link weights constitute a *bottleneck* metric, the *weight* $W(\mathcal{P})$ of a path $\mathcal{P}$ is defined as the weight of its worst link, *i.e.*, $W(\mathcal{P}) = \max_{e \in \mathcal{P}} w_e$.

We can define the notion of a path that is "best" when only path weights are considered. A *minimum-weight* (s, t)-path is a path $\mathcal{P} = \{s, ..., t\}$ whose weight is no larger than that of any other (s, t)-path.

Obviously, a minimum-weight path has the best performance with respect to the QoS requirement that is captured by the link weight metric; for instance, it is a path with minimum delay or maximum bandwidth. Minimum-weight paths can be efficiently found by Dijkstra's shortest-path algorithm, in $\mathcal{O}(M + N \log N)$ computational complexity [8]. Obviously, if the minimum-weight path fails to meet the connection's QoS requirement, then so does any other path. However, when the minimum-weight

path does meet the QoS requirement, it is often not the "right" choice, as it may be wasteful in terms of global network usage, *e.g.*, it may have a large number of hops, or it may use "expensive" links.

Therefore, the goal of QoS routing is to identify a path that satisfies a given QoS requirement while consuming as few resources as possible. Since the amount of the resources consumed on a path depends to a large extent on its number of links, the path hop count is considered to be a good criterion for estimating the path quality in terms of global resource utilization. When the hop count criterion is not satisfactory, one can define some *link cost* metric $c_e > 0$ that estimates the quality of each link e in terms of resource utilization; such a cost may depend on various factors, *e.g.*, the link's available bandwidth, its location, *etc*. The *cost* $C(\mathcal{P})$ of path $\mathcal{P}$ is defined to be the sum of the costs of its links, *i.e.*, $C(\mathcal{P}) = \sum_{e \in \mathcal{P}} c_e$.

In the present study we shall consider both cases of global utilization criteria, namely hop count and general (integer) link costs. Note that the former is a special case of the latter. For clarity, we say that a path $\mathcal{P}$ is a w-weight constrained if its weight is no more than w; similar definitions apply to h-hop constrained paths and c-cost constrained paths.

2.2 Problem Formulation

We are now ready to formulate the main problems that are considered in this study. Given a connection request between a source node $s \in V$ to a destination node $t \in V$ with a given QoS requirement $\hat{w}$, and given the network utilization preferences as captured by some link costs, the goal of the QoS routing scheme is to identify an (s,t)-path $\mathcal{P}$, which meets the QoS requirement at minimum cost. This can be formulated as a *restricted shortest path (RSP)* problem:

Problem RSP (Restricted Shortest Path) *Given are a source node s, a destination node t and a* QoS requirement $\hat{w}$*, find a minimum cost path $\hat{\mathcal{P}}$ between s and t such that $W(\hat{\mathcal{P}}) \leq \hat{w}$.*

We refer to a solution of Problem RSP as a *$\hat{w}$-weight constrained optimum (s,t)-path*.

For additive weights and general costs, Problem RSP is intractable, *i.e.*, $\mathcal{NP}$-hard [11]. However, there exist pseudo-polynomial solutions, based on dynamic programming, which give rise to fully polynomial approximation schemes (FPAS), whose computational complexity is reasonable (see, *e.g.*, [26] and references therein).

As mentioned above, many QoS routing problems consist of identifying, for each connection request, a path of minimum hop count that still meets the connection's bandwidth requirement. In other words, the path weight is a "bottleneck" metric, and its cost is equal to its number of hops. Effectively, these problems can be formulated as variants of Problem RSP, for which (i) weights are of the bottleneck type and (ii) links have equal costs; each of these two simplifications renders Problem RSP to be tractable. The first goal of this study is to provide efficient precomputation schemes for this class of problems, whose formal definition is presented next.

Problem BH-RSP (Bottleneck weight Hop cost RSP) *Given are a source node s, a destination node t and a bottleneck* QoS requirement $\hat{w}$*, find a path $\hat{\mathcal{P}}$ between s and t of minimum hop count such that $W(\hat{\mathcal{P}}) \leq \hat{w}$.*

As mentioned in the Introduction, QoS routing can often be considerably facilitated by means of employing a *precomputation scheme*, which performs the path search *a priori* for any possible connection request. Such a scheme comprises of two phases: the first phase prepares a database with precomputed paths for any possible QoS requirement; the second phase promptly retrieves the required path from the database upon a connection request.

Precomputation schemes for equal link costs (*i.e.*, minimum hops) were investigated in [1] and [13], both for bottleneck as well as additive weights. A simple precomputation scheme, which "inverts" the roles of the constraint (QoS requirement) and the optimization criterion (hops), was proposed in [1]. In particular, that scheme computes a minimum weight for each possible hop count; upon a connection

request, then, one would choose the minimum hop value for which the corresponding path meets the connection's QoS requirement. Accordingly, we define an *h-hop constrained optimal (s,t)-path* to be a path of minimum weight among all (s,t)-paths with hop count of at most h. The All Hops Optimum Path problem was then formulated in [13] as follows.

Problem AHOP (All Hops Optimal Path) *Given are a graph $G=(V,E)$, a source node $s \in V$ and a maximum hop count H. Find, for each hop value h, $1 \leq h \leq H$, and each destination node $t \in V$, an h-hop constrained optimal (s,t)-path.*

We shall contrast our precomputation scheme with schemes that are based on solving Problem AHOP. In addition, we shall consider the precomputation perspective in the context of *additive* QoS requirements and *general* path costs. Obviously, in this case, precomputation of exact solutions is intractable, since so is the basic underlying (RSP) problem. Therefore, in Section 5 we resort to precomputing *approximated* (ε-optimal) solutions.

3 Precomputation Scheme for Problem BH-RSP in Hierarchical Networks

In this section we present and analyze our precomputation scheme for the problem of hop minimization with "bottleneck" QoS constraints, *i.e.*, Problem BH-RSP, in hierarchical networks.

A possible approach for devising a precomputation scheme is to fully precompute all solutions during the first phase. With this approach, the second phase just consists of searching for the solution in the database produced by the first phase, according to the specific QoS requirement of the connection request. Such is indeed the precomputation scheme for Problem BH-RSP proposed in [13], which consists of precomputing all paths for all possible bandwidth requirements, *i.e.*, solving Problem AHOP. The Bellman-Ford shortest path algorithm provides a simple precomputation scheme for solving Problem AHOP, with a computational complexity of $\mathcal{O}(MH)$; for a general (dense) topology, that bound can grow to be as large as $\mathcal{O}(N^2H)$. In [13], an alternative scheme is described, whose computational complexity is $\mathcal{O}(N^2H/\log N)$; evidently, the latter outperforms Bellman-Ford's in dense topologies, *i.e.*, when $M > N^2/\log N$, but not in sparse topologies, which are the typical case of communication networks. The computational complexity incurred by the second phase of both schemes is just $\mathcal{O}(\log H + |\hat{\mathcal{P}}|)$, where $|\hat{\mathcal{P}}|$ is the hop count of the identified solution.

It thus remained an open question whether one can devise a faster precomputation scheme for typical network topologies. In the following, we demonstrate that, by exploiting the hierarchical structure that is typical of large-scale networks, one can establish a precomputation scheme for Problem BH-RSP, which offers a significant improvement upon the above solutions.

We begin with a formulation of the hierarchical model.

3.1 Hierarchical Model Formulation

A routing algorithm may be presented with a hierarchical topology due to two possible reasons. First, the (actual) topology intrinsically has a hierarchical structure, as is often the case with large-scale networks. Alternatively, the (actual) topology was hierarchically aggregated, as in the ATM PNNI recommendations [25].

We begin with the first case, assuming a *link state* routing environment, where the source node has a detailed image of the entire network. Beside being an interesting framework *per se*, it provides the required foundations for an extension that deals with the second case, which shall be considered in Section 4.

We assume that the network has a certain *hierarchical structure*. In order to state the precise meaning of the last term, we need to introduce some additional terminology.

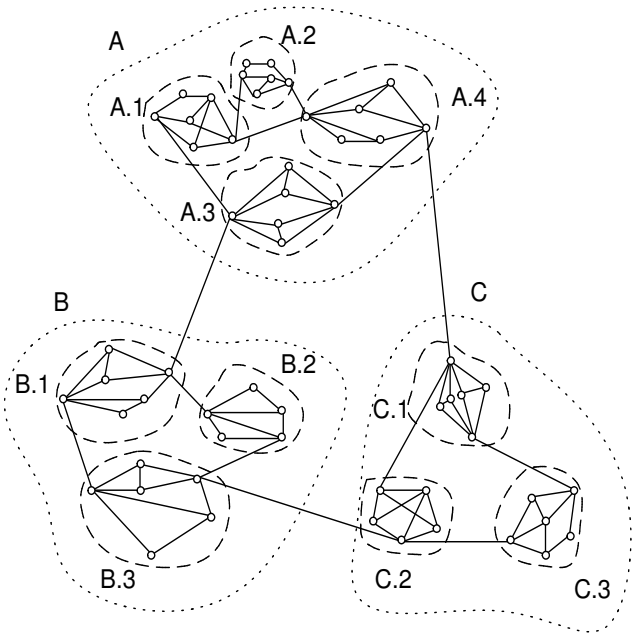


Figure 1: An example of a hierarchical network

The network $G = (V, E)$ is referred to as the *actual network*. Suppose that we partition the actual network nodes into some disjoint set of *layer*-1 *peer groups* (or *clusters*). Furthermore, suppose that we repeat the above process, such that, for each $k > 1$, layer-k peer groups are combined into layer-$(k+1)$ peer groups. We repeat this process until, for some K, we end up with a single layer-K peer group. Having performed such a (K-stage) partition, we say that nodes that form a layer-1 peer group Γ are its *children* and Γ is their *parent*. Similarly, layer-$(k-1)$ peer groups that form a layer-k peer group Γ are referred to as the children of Γ; Γ is referred to as their parent. We denote by H_Γ the maximum hop count of a path in a peer group Γ that can be considered for routing purposes. As we shall see, H_Γ plays an important role in our scheme.

A node in a peer group Γ, which has a neighbor that does not belong to Γ, is called a *border* node of Γ. For each peer group Γ that includes the source node s, we consider s to be a border node of Γ. We also assume that a path between two nodes of a peer group does not cross the peer group's boundary.

We are now ready to define the concept of *hierarchical structure*. Intuitively, it means that the network can be partitioned into peer groups, according to the above process, such that, for all k, $1 \leq k \leq K$, each layer-k peer group has a small number (at most d) of children, and, at the same time, the number of border nodes in a peer group is also small (at most b). Formally:

Definition 1 *A network $G(V, E)$ is said to be* (d, b)-hierarchical *if it can be iteratively clustered into some K layers of peer groups, according to the process described above, such that all the following hold:*

1. *For $1 \leq k \leq K$, each layer-k peer group is a union of at least 2 and at most d children.*
2. *The number of border nodes of each peer group is at most b.*

Note that, since each layer-$(k+1)$ peer group has at least 2 children, we have that $K = \mathcal{O}(\log M) = \mathcal{O}(\log N)$.

Let us illustrate the above terminology through an example. Fig. 1 depicts a $(6, 2)$-hierarchical network. $A.1, \cdots, A.4$,$B.1, \cdots, B.3$,$C.1, \cdots, C.3$ are layer-2 peer groups, while A, B, C are layer-3 peer groups. In this example, $K = 3$.

We assume that the hierarchical structure of the network, *i.e.*, partition into peer groups, is given. The next lemma establishes a "sparsity" property of hierarchical networks.

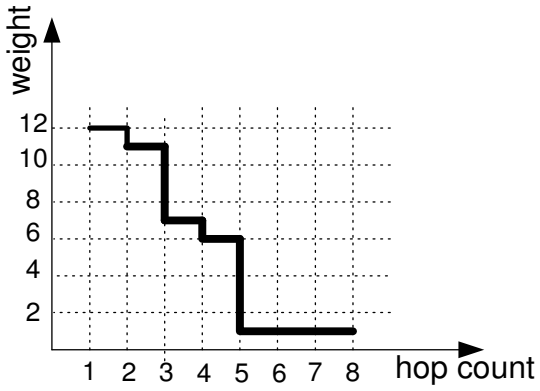


Figure 2: A typical traversal function $F^{\Gamma}_{(b_i,b_j)}(h)$

Lemma 1 *The number of links in a (d, b)-hierarchical network is at most $M = \mathcal{O}(b^2 dN)$.*

Proof: We divide the set E of actual network links into the following subsets $E_k, k = 0, ..., K-1$.

1. E_0 includes all links in E that connect nodes of the same layer-1 peer group;
2. E_k, for $k = 1, ..., K-1$, includes all links in E that connect layer-k peer groups.

Each node v is connected to at most d nodes of its layer-1 peer group. Thus, $|E_0| = \mathcal{O}(dN)$. Since each layer-k peer group comprises of at least 2 children, the number of layer-k peer groups is $\mathcal{O}(N/2^k)$. Note that each layer-k peer group, $k \geq 1$, is connected to at most d layer-k peer groups. Note also that each two layer-k peer groups are connected by at most b^2 links. Hence, for $k \geq 1$, it holds that $|E_k| = \mathcal{O}(b^2 dN/2^k)$. We conclude that $M = \sum_{k=0}^{K-1} |E_k| \leq \mathcal{O}(dN + b^2 d \sum_{k=1}^{K-1} N/2^k) = \mathcal{O}(b^2 dN)$ and the lemma follows. ■

In practical settings, d and b are small values. Typically, a network grows as a flat topology until its size reaches a certain threshold, which triggers the creation of a new peer group. As the network grows larger, more peer groups are added, but the size of each peer group remains bounded. Thus, the size of a peer group does not depend on the size of the network, hence we can assume that $d = \mathcal{O}(1)$. Border nodes connect a peer group to its neighbors; in typical settings, the number of neighbors does not depend on the network network size, *i.e.*, $b = \mathcal{O}(1)$. Hence, in such settings, we have $\mathcal{O}(b^2 d) = \mathcal{O}(1)$ and $M = \mathcal{O}(N)$.

3.2 Traversal functions

The precomputation scheme proposed in [13] consists of precomputing all paths for all possible bandwidth requirements. Our precomputation scheme is based on a different approach. Rather than explicitly precomputing a set of paths for each destination, our scheme computes *traversal functions* (defined below) for each peer group at each hierarchical layer. The traversal functions summarize the ability of a peer group to support QoS connections that may be established across it. Such an approach allows to exploit the network's hierarchical structure, and yields an efficient precomputation scheme. In addition, this approach is useful in networks with topology aggregation, as shall be shown in the next section.

Definition 2 *Given a peer group Γ and two border nodes, b_1 and b_2 of Γ, the* traversal function $F^{\Gamma}_{(b_1,b_2)}(h)$ *is defined to be the minimum weight of a (b_1, b_2)-path whose hop count is at most h.*

Algorithm BH-HIE (G, s)

input:

G- actual network;

$s \in G$- source node.

```
1  for k ← 1 to K−1
2    do for each layer-k peer group Γ
3         do for each border node b_i of Γ
4              do invoke Procedure CLUSTER for (G, Γ, b_i)
```

Figure 3: Algorithm BH-HIE

A typical traversal function $F^{\Gamma}_{(b_i,b_j)}(h)$ is depicted in Fig. 2. In this example, the minimum weight of a 5-hop constrained (b_i, b_j)-path across Γ is 1, while the minimum weight of 3-hop constrained path is 7.

3.3 First phase: Algorithm BH-HIE

We proceed to describe Algorithm BH-HIE, which implements the first phase of our precomputation scheme. For each peer group Γ and for each pair (b_i, b_j) of Γ's border nodes, the algorithm computes the corresponding traversal function $F^{\Gamma}_{(b_i,b_j)}(h)$.

The key idea is to compute the traversal function of a peer group out of the traversal functions of its childen. Accordingly, Algorithm BH-HIE runs across the hierarchical layers in a "bottom-up" manner, processing first peer groups of layer-1, then peer groups of layer-2, and so on, up the last, K's layer. For each peer group Γ and for each border node b_0 of Γ, the algorithm invokes Procedure CLUSTER, described in Section 3.3.1. The formal specification of Algorithm BH-HIE appears in Fig. 3.

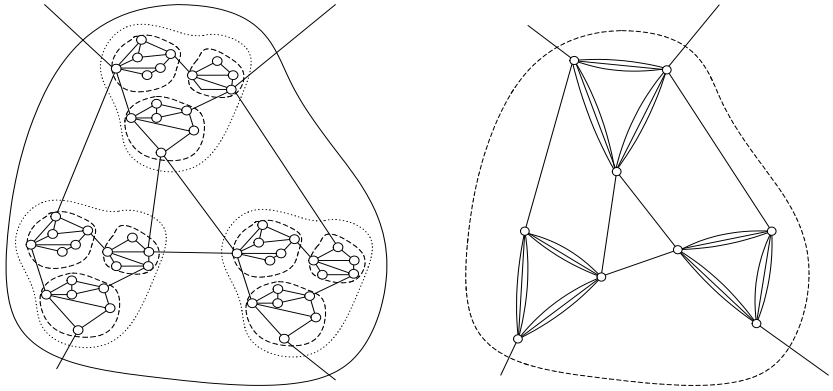
3.3.1 Procedure CLUSTER

We proceed to describe Procedure CLUSTER, the main building block of Algorithm BH-HIE. Procedure CLUSTER receives, as input, some layer-k peer group Γ and a node b_0, which is one of Γ's border nodes. It then (pre-)computes the traversal function $F^{\Gamma}_{(b_0,b)}(h)$ for each border node b of Γ. The order of processing of peer groups by Algorithm BH-HIE implies that, when Procedure CLUSTER is applied to a layer-k peer group Γ, it already has available the traversal functions $\left\{F^{\Gamma_i}_{(\cdot,\cdot)}(h)\right\}$ for each children Γ_i of Γ.

Since layer-1 peer groups have flat topologies we can employ the standard Bellman-Ford scheme. For all higher layers peer groups, however, we need more elaborate methods, in order to exploit their hierarchical structure.

The procedure starts by constructing the following auxiliary graph $\bar{\Gamma}(\bar{V}, \bar{E})$, whose purpose is to provide a concise representation of children and the connectivity among them. Each child Γ_i of Γ is represented in $\bar{\Gamma}$ by the set $\{b_{ij}\}$ of its border nodes. Each pair $(b_{ij}, b_{ij'})$ of Γ_i's border nodes is connected by several links, each corresponding to a different hop count constraint. Specifically, for each $h = 1, \cdots, H_{\Gamma_i}$, we add a link e between b_{ij} and $b_{ij'}$ in $\bar{\Gamma}$, with cost $c_e = h$ and weight $w_e = F^{\Gamma_i}_{(b_{ij},b_{ij'})}(h)$. In addition, for each actual network link $e(b_{ij}, b_{i'j'})$ that connects children of Γ, we add a link $e'(b_{ij}, b_{i'j'})$ to $\bar{\Gamma}$, with weight $w_{e'} = w_e$ and cost $c_{e'} = 1$. Fig. 4 illustrates the construction of the auxiliary graph for a layer-3 peer group.

The lemma below follows from the construction of $\bar{\Gamma}$ and from the validity of traversal functions of the children Γ_i of Γ.



(a) peer group Γ (b) auxiliary graph $\bar{\Gamma}$

Figure 4: Construction of auxiliary graph $\bar{\Gamma}$.

Lemma 2 *Let (b_i, b_j) be a pair of Γ's border nodes. Then:*

1. *For each (b_i, b_j)-path $\mathcal{P}$ in Γ, there exists a (b_i, b_j)-path $\bar{\mathcal{P}}$ in $\bar{\Gamma}$ such that $W(\bar{\mathcal{P}}) \leq W(\mathcal{P})$ and $C(\bar{\mathcal{P}}) \leq |\mathcal{P}|$.*
2. *For each (b_i, b_j)-path in $\bar{\Gamma}$, there exists a (b_i, b_j)-path $\mathcal{P}$ in Γ such that $W(\mathcal{P}) = W(\bar{\mathcal{P}})$ and $|\mathcal{P}| \leq C(\bar{\mathcal{P}})$.*

The lemma implies that we can use the auxiliary graph $\bar{\Gamma}$ for computing the traversal functions of Γ. Specifically, we need to find, for each border node b_i of Γ and for each $1 \leq c \leq H_\Gamma$, a minimum weight c-cost constrained (b_0, b_i)-path in $\bar{\Gamma}$. The key idea is to first remove (temporarily) all links from $\bar{\Gamma}$ and then add them back to $\bar{\Gamma}$ by increasing order of the weight values. During this process, we maintain the tree of minimum cost paths in $\bar{\Gamma}$ from the source node b_0; we update the tree after each insertion of a link to $\bar{\Gamma}$.

More specifically, for each $v \in \bar{\Gamma}$, we maintain the minimum cost c_v of a (b_0, v)-path in $\bar{\Gamma}$. In addition, we maintain an array $T^{\Gamma}_{(b_0,v)}$, such that:

- $T^{\Gamma}_{(b_0,v)}[c].w$ keeps the weight of a c-cost constrained optimum (b_0, v)-path $\hat{\mathcal{P}}$ in $\bar{\Gamma}$,
- $T^{\Gamma}_{(b_0,v)}[c].p$ keeps the predecessor u of v on $\hat{\mathcal{P}}$,
- $T^{\Gamma}_{(b_0,v)}[c].c$ keeps the cost of the last link (u, v) of $\hat{\mathcal{P}}$.

When we add a link $e(u, v)$ to $\bar{\Gamma}$, we check whether the value c_v decreases; if it does, we update c_v and set:

$$T^{\Gamma}_{(b_0,v)}[c_v].w \leftarrow w_e,$$
$$T^{\Gamma}_{(b_0,v)}[c_v].p \leftarrow u,$$
$$T^{\Gamma}_{(b_0,v)}[c_v].c \leftarrow c_e.$$

We perform a similar process for each node x for which c_x decreases as a result of adding link e. Upon completion, for each border node b_i of Γ, the traversal function $F^{\Gamma}_{(b_0,b_i)}(h)$ is stored in the array $T^{\Gamma}_{(b_0,b_i)}$, *i.e.*, for each h, $1 \leq h \leq H_\Gamma$:

$$F^{\Gamma}_{(b_0,b_i)}(h) = T^{\Gamma}_{(b_0,b_i)}[h].c.$$

The formal specification of Procedure CLUSTER appears in Fig. 5.

3.4 Procedure FIND

We proceed to present Procedure FIND. This procedure is invoked upon each new connection request, and its goal is to identify a minimum hop $\hat{w}$-weight constrained (s,t)-path $\hat{\mathcal{P}}$.

For clarity, we denote by Γ^1 the parent of t, by Γ^2 the parent of Γ^1, *etc*; up to some peer group $\Gamma^{\hat{k}}$ for which s is a border node. First, the procedure identifies, for each border node b_i of each peer group Γ^k, the minimum hop count $H_{(b_i,t)}$ of a $\hat{w}$-weight constrained optimal (b_i,t)-path in Γ^k. Next, a suitable path is determined by Procedure PATH, described in the Appendix A.

We begin with a layer-1 peer group Γ^1. We prune each link $e \in \Gamma^1$ for which $w_e > \hat{w}$, and then apply a Breadth First Search (BFS) algorithm [8] to the reversed graph, *i.e.*, a graph in which each link appears in the reverse direction. This yields, for each border node b_i of Γ^1, a $\hat{w}$-weight constrained optimal (b_i,t)-path in Γ^1 and its hop count $H_{(b_i,t)}$.

For a layer-k peer group Γ^k, $k \geq 2$, we construct the following auxiliary graph $\bar{\Gamma}^k$. Each child Γ_i of Γ^k is represented in $\bar{\Gamma}$ by the set $\{b_{ij}\}$ of its border nodes. Each pair $(b_{ij}, b_{ij'})$ of Γ_i's border nodes is connected a link e, whose cost is set to:

$$c_e \leftarrow \min\left\{h \;\middle|\; F^{\Gamma_i}_{(b_{ij},b_{ij'})}(h) \leq \hat{w}\right\}. \tag{1}$$

In addition, for each actual network link $e(b_{ij}, b_{i'j'})$ that connects children of Γ^k and whose weight is no more than $\hat{w}$, we add a link $e'(b_{ij}, b_{i'j'})$ to $\bar{\Gamma}$ with cost $c_e = 1$. Finally, we connect by a link each border node b_j of Γ^{k-1} and the destination node t. The cost of such a link is set to the $H_{(b_j,t)}$; the value of $H_{(b_j,t)}$ was computed in the previous iteration. We note that a minimum cost (b_i,b_j)-path in $\bar{\Gamma}^k$ corresponds to a $\hat{w}$-weight constrained path in the actual network. Having constructed the auxiliary graph $\bar{\Gamma}^k$, we identify, for each border node b_i of Γ^k, the minimum cost (b_i,t)-path $\bar{\mathcal{P}}^{\Gamma^k}(b_i,t)$ in $\bar{\Gamma}^k$, by applying Dijkstra's algorithm on the reverse graph of $\bar{\Gamma}^k$. Then, we set $H_{(b_i,t)} = C(\bar{\mathcal{P}}^{\Gamma^k}(b_i,t))$. In the last step, Procedure FIND invokes Procedure PATH that identifies, for a peer group Γ^k and a border node $b_i \in \Gamma^k$, a $\hat{w}$-weight constrained optimal (b_i,t)-path. The formal specification of Procedure FIND appears in Fig. 6.

3.5 Analysis of the precomputation scheme

First, we establish the following properties of Procedure CLUSTER.

Lemma 3 *Suppose that the (correct) traversal function $F^{\Gamma_i}_{(b_{ij},b_{ij'})}(h)$ is available for each pair $(b_{ij}, b_{ij'})$ of border nodes of each child Γ_i of Γ. Then, Procedure* CLUSTER*, applied on Γ and a border node b_0 of Γ, computes the traversal function $F^{\Gamma}_{(b_0,b_i)}(h)$ for each border node b_i of Γ.*

Proof: See Appendix B. ■

In the next lemma we analyze the complexity of Procedure CLUSTER.

Lemma 4 *The computational complexity of Procedure* CLUSTER *for a layer-k peer group is $\mathcal{O}(b^2d^2(H_\Gamma + \log(bd)))$.*

Proof: First, let us count the number of links in S. For each pair of border nodes of a child Γ_i of Γ we add at most $H_{\Gamma_i} \leq H_\Gamma$ links. Since each child has at most b border nodes and there are at most

Procedure CLUSTER (G, Γ, b_0)

input:
G- network
Γ- layer-k peer group
b_0 - Γ's border node

1 $\bar{\Gamma}(\bar{V}, \bar{E}) \leftarrow$INITIALIZE()
2 **for** all $v \in \bar{V}$ **do**
3 $c_v \leftarrow H_\Gamma + 1$
4 $c_{b_0} \leftarrow 0$
5 $T^{\Gamma}_{(b_0,b_0)}[0].w \leftarrow 0$, $T^{\Gamma}_{(b_0,b_0)}[0].p \leftarrow nil$
6 $S \leftarrow \bar{E}$
7 $\bar{E} \leftarrow \emptyset$
8 **for** each link $e(v,u) \in S$ by increasing order of w_e **do**
9 **if** exists a link e' between v and u in $\bar{E}$ **then**
10 remove e' from $\bar{E}$
11 add e to $\bar{E}$
12 PROPAGATE($e(v,u)$)
13 **return** $T^{\Gamma}_{(b_0,b)}$ for each border node b of Γ

PROPAGATE($e(u,v)$)
1 **if** $(c_u + c_e) < c_v$ **then**
2 **for** $c \leftarrow (c_u + c_e)$ **to** $(c_v - 1)$ **do**
3 $T^{\Gamma}_{(b_0,v)}[c].w \leftarrow w_e$, $T^{\Gamma}_{(b_0,v)}[c].p \leftarrow u$, $T^{\Gamma}_{(b_0,v)}[c].c \leftarrow c_e$
4 $c_v \leftarrow (c_u + c_e)$
5 **for** each link $e'(v,x) \in \bar{E}$ **do**
6 PROPAGATE($e'(v,x)$)

INITIALIZE()
1 $\bar{V} \leftarrow \emptyset$, $\bar{E} \leftarrow \emptyset$
2 **if** $k = 1$ **then**
3 **for** each link $e(v,u) \in \Gamma$ **do**
4 add a new link e' between v and u to $\bar{E}$
5 $c_{e'} \leftarrow 1$, $w_{e'} \leftarrow w_e$.
6 **else**
7 **for** each child Γ_i of Γ **do**
8 add each border node b_{ij} of Γ_i to $\bar{V}$.
9 **for** each pair $(b_{ij}, b_{ij'})$ of Γ_i's border nodes **do**
10 **for** $c \leftarrow 1$ **to** H_{Γ_i} **do**
11 add a new link e between b_{ij} and $b_{ij'}$ to $\bar{E}$
12 $c_e \leftarrow c$, $w_e \leftarrow F^{\Gamma_i}_{(b_{ij},b_{ij'})}(c)$
13 **for** each link $e(b_{ij}, b_{i'j'}) \in G$ that connects children Γ_i and $\Gamma_{i'}$ of Γ **do**
14 add a link e' between b_{ij} and $b_{i'j'}$ to $\bar{E}$
15 $c_{e'} \leftarrow 1$, $w_{e'} \leftarrow w_e$
16 **return** $\bar{\Gamma}(\bar{V}, \bar{E})$

Figure 5: Procedure CLUSTER

Procedure FIND ($G, s, t, \hat{w}$)

input:

$G(V, E)$- actual network
$s \in G$- source node
$t \in G$- destination node
$\hat{w}$- (bottleneck) QoS requirement

1 $\Gamma^1 \leftarrow$ parent of t
2 remove from Γ^1 all links e for which $w_e > \hat{w}$;
3 apply a Breadth First Search (BFS) algorithm [8] to the reversed graph $\bar{\Gamma}^1$ of Γ^1[a]
4 for each border node b_i of Γ^1 store the hop count of (b_i, t)-path in $\bar{\Gamma}^1$ computed in Line 3 into $H_{(b_i,t)}$
5 **for** $k \leftarrow 1$ **to** K **do**
6 $\Gamma^k \leftarrow$ parent of Γ^{k-1}
7 **for** each child Γ_i of Γ^k **do**
8 add each border node b_{ij} of Γ_i to $\bar{\Gamma}^k$
9 **for** each pair $(b_{ij}, b_{ij'})$ of Γ_i's border nodes **do**
10 add a new link e between b_{ij} and $b_{ij'}$ to $\bar{E}$
11 $c_e \leftarrow \min\{h \mid F^{\Gamma_i}_{(b_{ij},b_{ij'})}(h) \leq \hat{w}\}$
12 **for** each link $e = (b_{ij}, b_{i'j'}) \in G$ that connects children Γ_i and $\Gamma_{i'}$ of Γ^k **do**
13 **if** $w_e \leq \hat{w}$ **then**
14 add a link e' between b_{ij} and $b_{i'j'}$ to $\bar{\Gamma}^k$
15 $c_{e'} \leftarrow 1$
16 **for** each border node b_j of Γ^{k-1} **do**
17 add a link e between b_j and t
18 $c_e \leftarrow H_{(b_j,t)}$
19 apply Dijkstra's algorithm [8] for the reversed graph of $\bar{\Gamma}^k$
20 for each border node b_i of Γ^k store the cost of (b_i, t)-path in $\bar{\Gamma}^k$ computed in Line 19 into $H_{(b_i,t)}$
21 **if** s is a border node of Γ^k **then**
22 let $\bar{\mathcal{P}}$ be path returned by Dijkstra's algorithm
23 $\hat{\mathcal{P}} \leftarrow$PATH$(\Gamma^k, \bar{\mathcal{P}})$
24 **return** $\hat{\mathcal{P}}$

[a] $\bar{\Gamma}^1$ includes a link $\bar{e}(u, v)$ for each link $e(v, u) \in \Gamma^1$.

Figure 6: Procedure FIND

d children, the total number of such links is $\mathcal{O}(b^2 d H_\Gamma)$. In addition, we add $\mathcal{O}(b^2 d^2)$ links that connect border nodes of different children. Thus, the total number of links in S is $\mathcal{O}(b^2 d(H_\Gamma + d))$. Consequently, the complexity of Procedure INITIALIZE and of lines 2-11 of Procedure CLUSTER is $\mathcal{O}(b^2 d(H_\Gamma + d))$.

For sorting the elements of S we can use techniques for searching in arrays with sorted columns [10]. Since the elements of S are constructed from at most $b^2 d$ ordered sets and an additional set of at most $b^2 d^2$ elements, such a sorting can be performed in $\mathcal{O}(b^2 d(H_\Gamma + d)\log(bd))$ time.

We proceed to count the computational complexity incurred by recursive invocations of Procedure PROPAGATE. Note that it is sufficient to count invocations in which the condition $(c_v + c_e) < c_u$ is satisfied. In each invocation, c_u decreases by at least 1 for some $u \in \bar{\Gamma}$. Since c_u is initially set to $H_\Gamma + 1$ and never increases, we conclude that Procedure PROPAGATE is invoked at most $H_\Gamma + 1$ times for each node $u \in \bar{\Gamma}$. Each invocation of the procedure incurs $\mathcal{O}(d_{out}(u))$ steps, where $d_{out}(u)$ is the output degree of u in $\bar{\Gamma}$. Since during the execution of the algorithm, any two nodes of Γ are connected by at most one link, the number of links in $\bar{\Gamma}$ is $\mathcal{O}(b^2 d^2)$. Hence, the total running time for all invocations of Procedure PROPAGATE is at most $\mathcal{O}(H_\Gamma \sum_{u \in \bar{\Gamma}} d_{out}(u)) = \mathcal{O}(b^2 d^2 H_\Gamma)$.

We conclude that the total running time of Procedure CLUSTER is $\mathcal{O}(b^2 d^2 H_\Gamma + b^2 d \log(bd)(H_\Gamma + d))$=$\mathcal{O}(b^2 d^2 (H_\Gamma + \log(bd)))$. ■

We proceed to establish the following properties of Algorithm BH-HIE.

Theorem 1 *Algorithm* BH-HIE *determines, for each peer group* Γ *at each hierarchical layer, the traversal function* $F^{\Gamma}_{(b_i,b_j)}(h)$ *for each pair* (b_i, b_j) *of border nodes of* Γ.

Proof: Straightforward by induction on hierarchical layers and application of Lemma 3. ■

Lemma 5 *The computational complexity of Algorithm* BH-HIE *is* $\mathcal{O}(b^2 d^2 NK)$.

Proof: Let us count the time required to process all layer-k peer groups. For each layer-k peer group Γ_i, Procedure CLUSTER is invoked. By Lemma 4, the running time of Procedure CLUSTER is $\mathcal{O}(b^2 d^2 (H_{\Gamma_i} + \log(bd)))$. Since the expression $H_{\Gamma_i} + \log(bd)$ is upper bounded by the number of nodes in Γ_i and the total number of nodes in layer-k peer groups is N, processing all layer-k peer groups requires $\mathcal{O}(b^2 d^2 N)$ time. Since there are K hierarchical layers, the algorithm's complexity is $\mathcal{O}(b^2 d^2 KN)$. ■

Lemma 6 *Procedure* FIND, *invoked for a source* s, *destination* t *and QoS constraint* $\hat{w}$, *returns the minimum hop count of a* $\hat{w}$*-weight constrained* (s,t)*-path.*

Proof: See Appendix C. ■

We proceed to analyze the computational complexity of Procedure FIND.

Lemma 7 *The computational complexity of Procedure* FIND *is* $\mathcal{O}(b^2 dK(d + \log H) + |\hat{\mathcal{P}}|)$.

Proof: Note that, for each $1 \le k \le K$, the graph $\bar{\Gamma}^k$ contains at most $\mathcal{O}(bd)$ nodes and $\mathcal{O}(b^2 d^2)$ links. The execution of all lines in the procedure, except from lines 11 and 19, requires only a fixed number of steps per link, or $\mathcal{O}(b^2 d^2)$ per layer. Line 11 may be implemented in $\mathcal{O}(\log H)$ running time per link, by a binary search. This line is executed at most $\mathcal{O}(b^2 d)$ times for each layer, hence it incurs $\mathcal{O}(b^2 d \log H)$ steps per layer. Since the auxiliary graph $\hat{\Gamma}$ comprises of $\mathcal{O}(bd)$ nodes and $\mathcal{O}(b^2 d^2)$ links, the execution of Dijkstra's algorithm in line 19 requires $\mathcal{O}(b^2 d^2)$ time. As a result, the procedure performs $\mathcal{O}(b^2 d(d + \log H))$ operations per layer, or $\mathcal{O}(b^2 dK(d + \log H))$ operations overall. In addition, Procedure PATH needs $\mathcal{O}(|\hat{\mathcal{P}}|))$ time to report path $|\hat{\mathcal{P}}|$. ■

The above results are summarized in the following theorem.

Theorem 2 *Procedure* FIND *provides an* $\mathcal{O}(b^2 dK(d + \log H) + |\hat{\mathcal{P}}|)$ *solution to Problem BH-RSP,* i.e.*: given a connection request with source node* s, *destination node* t, *and (bottleneck) QoS constraint* $\hat{w}$, *and given the output of Algorithm* BH-HIE, *Procedure* FIND *identifies, in* $\mathcal{O}(b^2 dK(d + \log H) + |\hat{\mathcal{P}}|)$ *steps, a* $\hat{w}$*-weight optimal* (s,t)*-path* $\hat{\mathcal{P}}$ *in* G.

Note 1 *In some settings the detailed path is not required. For example, in order to decide whether to admit a connection, we only need to know the minimum hop count of a QoS path. Then, the computational complexity of Procedure* FIND *is just* $\mathcal{O}(b^2dK(d+\log H))$.

3.6 Discussion

We conclude this section with a performance comparison between our scheme and some alternatives.

Consider first the "standard" precomputation scheme proposed in [1, 13], which was based on solving Problem AHOP through Bellman-Ford's shortest path algorithm. Lemma 1 implies that hierarchical networks are sparse, in the sense that $M = \mathcal{O}(b^2dN)$. This, in turn, implies that the standard scheme incurs a computational complexity of $\mathcal{O}(b^2dNH)$ for its first phase, *i.e.*, it is $\Omega(H/(dK))$ times slower than ours. Since $K = \mathcal{O}(\log N)$, our scheme offers a significant improvement over the standard solution. The difference is particularly significant when $d = \mathcal{O}(1)$ and $K << H$,[1] which is a typical case.

Considering the second phase, the standard scheme (as well as any other which is based on fully solving Problem AHOP in the first phase) yields a computational complexity of just $\mathcal{O}(\log H + |\hat{\mathcal{P}}|)$, where $\hat{\mathcal{P}}$ is the path identified by the scheme. This is somewhat less than that of our scheme, *i.e.*, $\mathcal{O}(b^2dK(d+\log H) + |\hat{\mathcal{P}}|)$. However, in typical settings, where $b = \mathcal{O}(1)$, $d = \mathcal{O}(1)$ and $K << H$, our scheme is just $\Omega(\log H)$ times slower than the standard solution. Moreover, the difference is not significant when $|\hat{\mathcal{P}}|$ is the dominating component.

Next, let us compare between our precomputation scheme and an alternative where no precomputation is performed at all. In such a "single-phase" scheme, the required path can be identified by applying a Breadth-First Search algorithm [8], which, for $M = \mathcal{O}(b^2dN)$, incurs $\mathcal{O}(b^2dN)$ running time compared to $\mathcal{O}(b^2dK(d+\log H) + |\hat{\mathcal{P}}|)$ of our scheme. Since $K = \mathcal{O}(\log N)$, our solution is $\Omega(N/(\log N(d+\log H) + |\hat{\mathcal{P}}|))$ times faster. Typically, $|\hat{\mathcal{P}}| << N$ and $d = \mathcal{O}(1)$, hence the difference is significant.

It is interesting to compare between the two approaches also in the related context of *connection admission*, where one needs to decide whether a connection request should be admitted, based on its QoS requirement and the cost it incurs; to that end, one needs to identify the (best) cost of a path over which the connection can be established, however there is no need to explicitly specify the path itself. This means that our scheme allows to obtain an admission decision upon a connection request in just $\mathcal{O}(b^2dK(d+\log H))$ time (see Note 1), whereas the "single-phase" scheme still incurs $\mathcal{O}(b^2dN)$ time. Thus, our solution is $\Omega((N/(\log N(d+\log H)))$ times faster. The difference is significant in typical settings where $d = \mathcal{O}(1)$.

4 Precomputation Scheme for Aggregated Networks

In the previous section we assumed that each node has full and unabridged information about link states, which is stored in its topology database. However, such an approach suffers from scalability problems. Indeed, as the network grows in size, significant resources are consumed for flooding and recording the changes in the link state throughout the network. With topology aggregation, subnetworks, or *routing domains*, do not reveal the details of their internal structure, but rather supply the aggregated representation to the outside world [16]. Such an approach could also be mandated by security and administrative needs. Topology aggregation is useful for ATM [25] and IP networks [21].

A key issue in topology aggregation is how to provide the routing information of a domain to the outside world. Constructing an accurate *aggregated representation* poses several complex challenges. First, there is a need to identify the *traversal characteristics* of a routing domain, *i.e.*, its ability to support

[1] Recall that H is the maximum hop count of a path in G and, in the worst case, $H = \mathcal{O}(N)$, while $K = \mathcal{O}(\log N)$.

connections with different QoS requirements. Second, each domain, in turn, comprises of aggregated sub-domains, whose traversal characteristics are available only through aggregated representation. Finally, each sub-domain may use a different method for representing its routing information.

In this section we establish that precomputation is a useful (virtually necessary) tool for constructing accurate aggregated representations. In particular, we show that, by using precomputation schemes, the traversal characteristics of a peer group can be computed efficiently. We employ the concept of *traversal functions*, introduced in Section 3, in order to accurately represent the traversal characteristics. Further, we adapt our algorithmic techniques in order to cope with aggregated networks, *i.e.*, networks in which each link represents an aggregated sub-domain.

In this section we consider a generic model for multi-level aggregated networks. The model can be used, for example, in conjunction with the ATM PNNI protocol [25], in which peer groups are represented by structures termed *complex nodes*.[2]

Similarly to the previous section, we focus on bottleneck QoS requirements and use hop minimization for optimizing resource utilization.

4.1 Aggregated Representation of a Peer Group

A significant body of research has been devoted to the area of topology aggregation; we proceed to quote a few relevant references. A compact $\mathcal{O}(b)$ representation for undirected networks and a single bottleneck QoS requirement was presented in [16]. For additive QoS constraints, [23] shows that an accurate representation of a peer group requires $\mathcal{O}(b^2)$ links in the worst case. An $\mathcal{O}(b)$ representation that achieves a bounded distortion is presented in [4].

Devising a topology aggregation scheme that takes into account both the QoS constraints and the use of network resources is still an open research issue. A practical approach is to associate each pair of border nodes with two values: the first corresponds to a (bottleneck) weight and the second to a cost value (*e.g.*, hop count). This approach provides a compact representation, but suffers from high distortion. In order to reduce distortion, some studies [5, 15, 19] present schemes that resemble the traversal functions introduced in Section 3. Specifically, [5] proposes to use bandwidth-cost functions, *i.e.*, functions that specify the available bandwidth for several cost values; [15] and [19] propose to approximate the bandwidth-cost functions by using curves and link segments, respectively. We note that the methods presented in the following for efficiently computing traversal functions can be used in order to compute the curves in [15] and line segments in [19], hence resulting in a more efficient scheme.

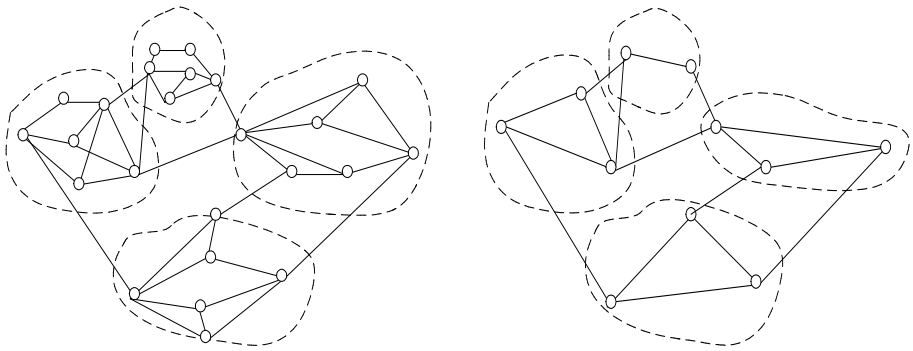
4.2 Aggregated Model

In multi-level aggregated topologies, a domain comprises of aggregated sub-domains. This gives rise to the following *aggregated network model*.

The actual network topology (*i.e.*, with no aggregation) is represented by a directed graph G. The aggregated network is represented by a directed graph $\hat{G}$, in which each link $e(v,u) \in \hat{G}$ represents an aggregated sub-domain Γ^e of G. Fig. 7 depicts an example of an actual network and its corresponding aggregated topology. We assume that a node only knows the aggregated topology $\hat{G}$. We denote by H the maximum hop count of a path that can be considered for routing purposes in the actual network G. Each link $e(v,u) \in \hat{G}$ is associated with a traversal function $F_{(v,u)}(c)$, which provides the minimum weight value that can be supported by Γ^e for each hop count limitation c. More specifically, for $0 \leq c \leq H_{\Gamma_e}$, $F_{(v,u)}(c)$ is the minimum weight of a c-hop constrained (v,u)-path across the sub-domain Γ_e, where H_{Γ_e} is a maximum hop count of a path across Γ_e that can be considered for routing purposes.

Each path $\hat{\mathcal{P}}$ in $\hat{G}$ can support several QoS constraints at different costs. Accordingly, we define the cost $C(\hat{\mathcal{P}}, \hat{w})$ of supporting the QoS constraint $\hat{w}$ by $\hat{\mathcal{P}}$.

[2] This requires a mild extension of the complex node structure. Specifically, we allow parallel bypass links, each link corresponds to a different value of the QoS constraint.



(a) actual network (b) aggregated network

Figure 7: Actual and aggregated networks.

Definition 3 *Given a path $\hat{\mathcal{P}} = \{s = v_0, v_1, \cdots, v_h = t\}$ in $\hat{G}$ and a QoS constraint $\hat{w}$, we define, for each link $e(v_{i-1}, v_i) \in \hat{\mathcal{P}}$, the* local cost *of supporting $\hat{w}$ to be $c_{(e,\hat{w})} = \min\{c \mid F_{(v_{i-1},v_i)}(c) \leq \hat{w}\}$. The cost of satisfying the QoS constraint $\hat{w}$ along the path $\hat{\mathcal{P}}$ is then defined to be:*

$$C(\hat{\mathcal{P}}, \hat{w}) = \sum_{e \in \hat{\mathcal{P}}} c_{(e,\hat{w})}.$$

A traversal function in an aggregated network is defined as follows:

Definition 4 *Given an aggregated network $\hat{G}$, a source node $s \in \hat{G}$ and a destination node $t \in \hat{G}$, we define the* aggregated traversal function $F^{\hat{G}}_{(s,t)}(c)$, $1 \leq c \leq H$, *to be the minimum weight w of an (s,t)-path $\mathcal{P}$ in $\hat{G}$ for which $C(\mathcal{P}, w) \leq c$. If no such path exists, $F^{\hat{G}}_{(s,t)}(c)$ is defined as ∞.*

Intuitively, the aggregated traversal function in $\hat{G}$ is identical to the traversal function in the actual network G. Furthermore, the traversal function $F_{(v,u)}(c)$ that is associated with each link $(v,u) \in \hat{G}$ is, in fact, the aggregated traversal function of the sub-domain Γ^e. Note that Γ^e may, in turn, comprise of aggregated sub-domains.

We proceed to formulate the aggregated version of Problem AHOP.

Problem Agg-AHOP (Aggregated All Hops Optimal Path) *Let $\hat{G}$ be an aggregated network, where each link $e(v,u) \in \hat{G}$ is associated with a traversal function $F_{(v,u)}(c)$. For a source node $s \in \hat{G}$ and each destination node $t \in \hat{G}$, find the aggregated traversal function $F^{\hat{G}}_{(s,t)}(c)$.*

Problem Agg-AHOP can be solved by substituting each link $e \in \hat{G}$ with several links, each link (v,u) being associated with a single weight w and cost value c, such that $F_{(v,u)}(c) = w$, and then applying the Bellman-Ford algorithm on the resulting graph. However, as the resulting graph includes $\mathcal{O}(MH)$ links, this approach incurs a high computational complexity of $\mathcal{O}(MH^2)$. By using the algorithmic methods developed in the previous section, we can devise an alternative scheme for Problem Agg-AHOP, whose computational complexity is significantly lower.

4.3 Precomputation Scheme

Consider first a simple case, in which $\hat{G}$ comprises of just links (u, v) and (v, w), and our goal is to compute a traversal function $F^{\hat{G}}_{(v,w)}(c)$. We refer to this operation as *merging* the functions $F_{(v,u)}(c)$ and $F_{(u,w)}(c)$ into a single function $F^{\hat{G}}_{(v,w)}(c)$. The merge operation essentially amounts to computing, for each budget c, $1 \leq c \leq H$, the partition (c^1, c^2) of the budget between the links (u, v) and (v, w) that minimizes the weight of a (u, w)-path in the actual network, *i.e.*,

$$F^{\hat{G}}_{(v,w)}(c) = \min_{c^1+c^2 \leq c} \left\{ \max \left\{ F_{(u,v)}(c^1), F_{(v,w)}(c^2) \right\} \right\}.$$

Our main observation is that, in the case of bottleneck QoS parameters, the merge operation can be performed in just $\mathcal{O}(H)$ steps, through the following inductive process. Clearly, for budget $c_0 = 2$, the optimal partition is $(1, 1)$. Having computed the optimal partition (c^1_{i-1}, c^2_{i-1}) for a budget c_{i-1}, the optimal partition for a budget $c_i = c_{i-1} + 1$ is then either $(c^1_{i-1} + 1, c^2_{i-1})$ or $(c^1_{i-1}, c^2_{i-1} + 1)$:

$$F_{(u,w)}(c_i) = \min \left\{ \max \left\{ F_{(u,v)}(c^1_{i-1} + 1), F_{(v,w)}(c^2_{i-1}) \right\} , \right.$$
$$\left. \max \left\{ F_{(u,v)}(c^1_{i-1}), F_{(v,w)}(c^2_{i-1} + 1) \right\} \right\}.$$

The merge operation allows to solve Problem Agg-AHOP in acyclic directed graphs in $\mathcal{O}(MH)$ time. For general directed graphs, we present a more elaborated algorithm that utilizes that same idea, and whose running time is $\mathcal{O}(MH \log N)$. The algorithm, referred to as Algorithm AGG-AHOP, is, in fact, an adaptation of Procedure CLUSTER (Section 3) for networks with topology aggregation.

The algorithm starts by constructing the following auxiliary graph $\bar{G}(\bar{V}, \bar{E})$, whose purpose is to represent traversal characteristics of sub-domains and the connectivity among them. Each link $e(v, u)$ of $\hat{G}$ is represented in $\bar{G}$ by several links, each corresponding to a different cost constraint. Specifically, for each $c = 1, \cdots, H_{\Gamma^e}$, we add a link e' between v and u in $\bar{G}$, with cost $c_{e'} = c$ and weight $w_{e'} = F_{(v,u)}(c)$, where Γ^e is the aggregated sub-domain represented by e. For each $v \in \bar{G}$, we maintain the minimum cost c_v of an (s, v)-path in $\bar{G}$. In addition, we maintain array $T^{\hat{G}}_{(s,v)}$, such that:

- $T^{\hat{G}}_{(s,v)}[c].w$ keeps the minimum weight of a c-cost constrained (s, v)-path $\hat{\mathcal{P}}$ in $\bar{G}$,
- $T^{\hat{G}}_{(s,v)}[c].p$ keeps the predecessor of v on $\hat{\mathcal{P}}$,
- $T^{\Gamma}_{(b_0,v)}[c].c$ keeps the cost of the last link (u, v) of $\hat{\mathcal{P}}$.

The key idea is to first remove (temporary) all links from $\bar{G}$ and then add them back to $\bar{G}$ by increasing order of the weight values. When we add a link $e(u, v)$ to $\bar{G}$, we check whether the value c_v decreases; and, if it does, we update c_v and set

$$T^{\hat{G}}_{(s,v)}[c_v].w \leftarrow w_e, T^{\hat{G}}_{(s,v)}[c_v].p \leftarrow u, T^{\Gamma}_{(s,v)}[c_v].c \leftarrow c_e.$$

We perform a similar process for each node x for which c_x decreases as a result of adding link e. Upon completion, for each node v of Γ, the traversal function $F^{\hat{G}}_{(s,v)}(c)$ is stored in the array $T^{\hat{G}}_{(s,v)}[c]$. The formal specification of Algorithm AGG-AHOP appears in Fig. 8.

Given a QoS constraint $\hat{w}$ and a destination node t, we determine a suitable path $\hat{\mathcal{P}}$ through the following procedure. First, we determine the minimum cost $\hat{c}$ of a (s, t)-path is $\hat{G}$ that supports $\hat{w}$ by setting $\hat{c} = \min \left\{ c \;\middle|\; T^{\hat{G}}_{(s,t)}[c].w \leq \hat{w} \right\}$. Next, path $\hat{\mathcal{P}}$ is identified by iteratively discovering the predecessor of each node, beginning with t. The predecessor v_{i-1} of v_i is determined by setting

Algorithm AGG-AHOP (G, s)

input:

$\hat{G}$- aggregated network

s - source node

1 $\bar{G}(\bar{V}, \bar{E}) \leftarrow$INITIALIZE()
2 **for** all $v \in \bar{V}$ **do**
3 $c_v \leftarrow H + 1$
4 $c_s \leftarrow 0$
5 $T^{\hat{G}}_{(s,s)}[0].w \leftarrow 0$, $T^{\hat{G}}_{(s,s)}[0].p \leftarrow nil$
6 $S \leftarrow \bar{E}$
7 $\bar{E} \leftarrow \emptyset$
8 **for** each link $e(v, u) \in S$ by increasing order of w_e **do**
9 **if** exists a link e' between v and u in $\bar{E}$ **then**
10 remove e' from $\bar{E}$
11 add e to $\bar{E}$
12 PROPAGATE$(e(v, u))$
13 **return** $T^{\hat{G}}_{(s,v)}$ for each node v in $\hat{G}$

PROPAGATE($e(u, v)$)

1 **if** $(c_u + c_e) < c_v$ **then**
2 **for** $c \leftarrow (c_u + c_e)$ **to** $(c_v - 1)$ **do**
3 $T^{\hat{G}}_{(s,v)}[c].w \leftarrow w_e$, $T^{\hat{G}}_{(s,v)}[c].p \leftarrow u$, $T^{\hat{G}}_{(s,v)}[c].c \leftarrow c_e$
4 $c_v \leftarrow (c_u + c_e)$
5 **for** each $e'(v, x) \in \bar{E}$ **do**
6 PROPAGATE($e'(v, x)$)

INITIALIZE()

1 $\bar{V} \leftarrow V$, $\bar{E} \leftarrow \emptyset$
2 **for** each $e = (v, u) \in \hat{G}$ **do**
3 **for** $c \leftarrow 1$ **to** H_{Γ^e} **do**
4 add a new link e' between v and u to $\bar{E}$
5 $c_{e'} \leftarrow c$, $w_{e'} \leftarrow F_{(v,u)}(c)$
6 **return** $\bar{G}(\bar{V}, \bar{E})$

Figure 8: Algorithm AGG-AHOP

$v_{i-1} = T^{\hat{G}}_{(s,v_i)}[\hat{c} - C(\hat{\mathcal{P}}_{(v_i,t)}, \hat{w})].p$, where $C(\hat{\mathcal{P}}_{(v_i,t)}, \hat{w})$ is the cost of supporting $\hat{w}$ by the subpath of $\hat{\mathcal{P}}$ identified so far. The budget c_e allocated to link $e(v_{i-1}, v_i)$ of $\hat{\mathcal{P}}$ is set to $c_e = T^{\hat{G}}_{(s,v_i)}[\hat{c} - C(\hat{\mathcal{P}}_{(v_i,t)}, \hat{w})].c$. We note that the identification of $\hat{\mathcal{P}}$ requires $\mathcal{O}(\log H + |\hat{\mathcal{P}}|)$ time.

4.4 Analysis of the precomputation scheme.

We proceed to state the following properties of Algorithm AGG-AHOP.

Lemma 8 *Algorithm* AGG-AHOP *computes the aggregate traversal function* $F^{\hat{G}}_{(s,t)}(c)$ *for each node* t *of* $\hat{G}$.

Proof: See Appendix D. ■

Lemma 9 *The computational complexity of Algorithm* AGG-AHOP *is* $\mathcal{O}(MH \log N)$.

Proof: First, let us count the number of links in S. For link $e \in \hat{G}$ we add at most $H_{\Gamma^e} \leq H$ links. Thus, the total number of links in S is $\mathcal{O}(MH)$. Consequently, the complexity of Procedure INITIALIZE and of lines 2-11 of Algorithm AGG-AHOP is $\mathcal{O}(MH)$.

For sorting the elements of S we can use techniques for searching in arrays with sorted columns [10]. Since the elements of S are constructed from at most M ordered sets, such a sorting can be performed in $\mathcal{O}(MH \log N)$ time.

We proceed to count the computational complexity incurred by recursive invocations of Procedure PROPAGATE. Note that it is sufficient to count invocations in which the condition $(c_v + c_e) < c_u$ is satisfied. In each invocation, c_u decreases by at least 1 for some $u \in \bar{\Gamma}$. Since c_u is initially set to $H + 1$ and never increases, we conclude that Procedure PROPAGATE is invoked at most $H + 1$ times for each node $u \in \bar{\Gamma}$. Each invocation of the procedure incurs $\mathcal{O}(d_{out}(u))$ steps, where $d_{out}(u)$ is the output degree of u in $\bar{G}$. Since during the execution of the algorithm, any two nodes of Γ are connected by at most one link, the number of links in $\bar{G}$ is $\mathcal{O}(M)$. Hence, the total running time for all invocations of Procedure PROPAGATE is at most $\mathcal{O}(H \sum_{u \in \bar{G}} d_{out}(u)) = \mathcal{O}(MH)$.

We conclude that the total running time of Algorithm AGG-AHOP is $\mathcal{O}(MH \log N)$ and the lemma follows. ■

The above results are summarized in the following theorem.

Theorem 3 *Algorithm* AGG-AHOP *determines, in* $\mathcal{O}(MH \log N)$ *time the aggregate traversal function* $F^{\hat{G}}_{(s,t)}(c)$ *for each node* $t \in \hat{G}$.

4.5 Discussion

We presented an $\mathcal{O}(MH \log N)$ algorithm for computing traversal functions in an aggregated environment. As previously noted, a straightforward approach would be to substitute each link by $\mathcal{O}(H)$ links and execute Bellman-Ford algorithm in the resulting graph. Since the Bellman-Ford algorithm would then be applied to a graph with $\mathcal{O}(MH)$ links, its computational complexity would be $\mathcal{O}(MH^2)$, which is $\Omega(H/\log N)$ times higher than that of our scheme. Recall that H is the maximum hop count in the *actual* network, whereas N is the number of nodes in the *aggregated* network, hence our improvement is significant.

Next, let us compare between our precomputation scheme and an alternative where no precomputation is performed at all. In such a "single-phase" scheme, the required path can be identified by computing, for each link $e(v,u) \in \hat{G}$, the cost c_e of supporting the QoS constraint $\hat{w}$ (*i.e.*, $c_e = \min\{c \mid F_{(v,u)}(c) \leq \hat{w}\}$), and then applying Dijkstra's shortest path algorithm [8] to a graph with link costs c_e. This scheme incurs $\mathcal{O}(M \log H + N \log N)$ running time, compared to $\mathcal{O}(\log H + |\hat{\mathcal{P}}|)$ in our scheme. Hence, our scheme allows to significantly reduce the time required for the identification of a suitable path.

5 Precomputation Schemes for Additive Metrics

In this section we consider the routing problem with *additive* QoS constraints and *general* links costs. We assume a link state environment, *i.e.*, the source node has a full image of the network. We consider general networks, *i.e.*, we do not assume that the network has a specific (*e.g.*, hierarchical) structure. Our purpose is to devise a scheme that (pre)computes, for each cost $0 \leq c \leq C$ and for each destination node $t \in G$, a c-cost constrained (s,t)-path of minimum weight, where C is maximal cost of a path that can be considered for routing purposes. Accordingly, we introduce Problem ACOP, which is a generalization of Problem AHOP for general link costs.

Problem ACOP (All Costs Optimal Path) *Given are a graph $G = (V, E)$, a source node $s \in V$ and a maximum cost C. Find, for each cost c, $1 \leq c \leq C$, and each destination node $t \in G$, a c-cost constrained (s,t)-path of minimum weight.*

Problem ACOP is computationally intractable since it contains Problem RSP, which is $\mathcal{NP}$-hard. Accordingly, we resort to precomputation schemes that offer *approximate* solutions, *i.e.*:

Definition 5 *Given an instance of Problem ACOP, with source node s, maximum cost C, and approximate ratio ε, $0 < \varepsilon \leq 1$, an* ε-approximate solution *is a set of paths S, such that, for each $0 \leq c \leq C$ and $t \in G$, there exists an (s,t)-path $\hat{\mathcal{P}} \in S$ that satisfies :*

1. *$W(\hat{\mathcal{P}}) \leq W(\mathcal{P})$, for any c-cost constrained (s,t)-path $\mathcal{P}$;*
2. *$C(\hat{\mathcal{P}}) \leq (1+\varepsilon)c$.*

We note that an approximate solution for Problem ACOP can be constructed on the basis of existing approximation algorithms for Problem RSP (*e.g.*, [14], [26]), *i.e.*, by sequentially executing them for various values of the cost constraint. However, as we shall see, such a simplistic approach results in a (overly) high computational complexity. Therefore, in this section we propose a scheme that precomputes a set of suitable paths within $\mathcal{O}(\frac{1}{\varepsilon}HM\log C)$ computational complexity. Upon a connection request, a suitable path is chosen from a set of precomputed path within $\mathcal{O}(\log(\frac{1}{\varepsilon}H\log C))$ time.

The section is organized as follows. First, we present a simple precomputation scheme whose running time is $\mathcal{O}(MC)$, which is pseudo-polynomial. Next, by using a *logarithmic scaling* technique, we establish a $\mathcal{O}(\frac{1}{\varepsilon}HM\log C)$ precomputation scheme that offers an ε-approximate solution for Problem ACOP.

5.1 Pseudo-polynomial Solution for Problem RSP

As a first step, we present a simple precomputation scheme, whose computational complexity is pseudo-polynomial. The scheme is based on dynamic programming and is an extension of the standard Bellman-Ford's algorithm. For each node $v \in G$ we maintain array $T_{(s,v)}[c]$ such that $T_{(s,v)}[c].w$ keeps the minimum weight of a c-cost constrained (s,v)-path in G and $T_{(s,v)}[c].p$ keeps the predecessor of v in that path. The algorithm iterates over "budget" values $c = 0, 1, \cdots, C$. At each iteration, the algorithm repeatedly selects a link $e \in G$ and relaxes it. The process of relaxing a link $e(v,u)$ consists of testing whether the minimum weight of (v,u)-path can be improved by going through v under the current budget restriction c and, if so, updating $T_{(s,u)}[c]$. Since for each $c, 1 \leq c \leq C$, the algorithm performs $\mathcal{O}(M)$ operations, its complexity is $\mathcal{O}(MC)$. The formal specification of Algorithm PP-RSP appears in Fig. 9.

Upon arrival of a connection request for a $\hat{w}$-weight constrained optimal (s,t)-path $\hat{\mathcal{P}}$, we first find the minimum cost $\hat{c}$ of $\hat{\mathcal{P}}$ by setting $\hat{c} = \min\left\{c |\ T_{(s,t)}[c].w \leq \hat{w}\right\}$. Then, we identify the path $\hat{\mathcal{P}}$ by using the information stored in the arrays $\{T_{(s,v)}[c] | v \in G\}$. Specifically, the predecessor v_{h-1} of t in $\hat{\mathcal{P}}$ is determined by setting $v_{h-1} = T_{(s,t)}[\hat{c}].p$. Generally, the predecessor v_{i-1} of v_i is determined by setting $v_{i-1} = T_{(s,v_i)}\left[\hat{c} - C(\hat{\mathcal{P}}_{(v_i,t)})\right].p$, where $\hat{\mathcal{P}}_{(v_i,t)}$ is the subpath of $\hat{\mathcal{P}}$, discovered so far. The algorithm outputs the resulting path $\hat{\mathcal{P}} = \{s = v_0, \cdots, v_h = t\}$.

Algorithm PP-RSP ($G(V,E), s, C$**)**

input:
$G(V,E)$ - network
$s \in G$ - source node
C- the maximum cost of a path
variables:
c - the "budget";
for all $v \in V$
$T_{(s,v)}[c]$ - auxiliary array

```
1  for all v ∈ V do
2    T_(s,v)[0].w ← ∞
3  T_(s,s)[0].w ← 0, T_(s,s)[0].p ← nil
4  for c ← 1 to C do
5    for each v ∈ V do
6      T_(s,v)[c].w ← T_(s,v)[c − 1].w
7      T_(s,v)[c].p ← T_(s,v)[c − 1].p
8    for each link e(v, u) ∈ E do
9      if (c_e ≤ c) then
10       if T_(s,v)[c − c_e].w + w_e ≤ T_(s,u)[c].w then
11         T_(s,u)[c].w ← T_(s,v)[c − c_e].w + w_e
12         T_(s,u)[c].p ← v
```

Figure 9: Algorithm PP-RSP

5.2 Polynomial Precomputation (Approximation) Scheme

We proceed to present an efficient precomputation scheme that provides an ε-approximate solution to Problem ACOP. The scheme is based on the pseudo-polynomial solution and uses a *logarithmic scaling* approach. Specifically, it considers only a limited number of budget values, namely $1, c_1, c_2, \cdots, c_{i_{max}}$, where $c_i = \delta^i$, $i_{max} = \min\{i \mid \delta^i \geq C\}$ and $\delta = (1 + \frac{\varepsilon}{6H})$. For each node $v \in G$ we maintain array $T_{(s,v)}[c]$ such that, for $0 \leq i \leq i_{max}$, $T_{(s,v)}[c_i].w$ keeps the minimum weight of a c_i-cost constrained (s,v)-path in G and $T_{(s,v)}[c_i].p$ keeps the predecessor of v in that path. The algorithm iterates over "budget" values $c_i = 1, c_1, c_2, \cdots, c_{i_{max}}$. At each iteration, the algorithm repeatedly selects a link $e \in G$ and relaxes it. The process of relaxing a link (v,u) consists of testing whether the minimum weight of (v,u)-path can be improved by going through v under the current budget restriction c_i and, if so, updating $T_{(s,u)}[c_i]$. As shall be shown below, the set of such paths constitutes a ε-optimal solution for Problem ACOP. The formal specification of Algorithm RSP-GEN appears in Fig. 10.

We will demonstrate the precomputation process by using the network G depicted on Fig. 11. Algorithm RSP-GEN is invoked for G,s, $\varepsilon = 1$ and $H = 5$. Thus, $\delta = 1.1$. We consider a request for a (s,t)-path that satisfies a QoS constraint $\hat{w} = 15$. For this request, $\hat{\mathcal{P}} = \{s, v_1, v_2, t\} \in G$ is an optimal path. We show that the algorithm identifies a path whose weight is at most $\hat{w}$ and whose cost is at most $(1+\varepsilon)C(\hat{\mathcal{P}})$. First, after executing line 4, we have $T_{(s,s)}[0].w = 0$. Next, consider the execution of the main loop, *i.e.*, the loop that begins at line 6, for $i = 12$. Since $\delta^i = 3.138 > 3$, the condition of line 12 is satisfied, hence, upon completion of the iteration, it holds that $T_{(s,v_1)}[3.138].w \leq 2$. Next, consider the iteration of the main loop for $i = 23$ and the iteration of the sub-loop at line 11 for $e = (v_1, v_2)$. In line 13 we set $c = 3.797$, which is the highest degree of δ that is lower than $c_i - c_e = 3.954$, where $c_i = \delta^i = 8.945$. In the next lines we check whether $T_{(s,v_1)}[3.797].w + w_e \leq T_{(s,v_2)}[8.954].w$ and, if so, we assign $T_{(s,v_2)}[8.954].w = T_{(s,v_1)}[3.945].w + w_e$. Thus, after completion of the iteration of the main loop for $i = 23$, we have $T_{(s,v_2)}[8.954].w \leq 9$. Finally after the completion of the iteration for

Algorithm RSP-GEN ($G(V,E), s, \varepsilon, C$**)**

input:
$G(V,E)$ - network
$s \in G$ - source node
ε, $0 < \varepsilon \leq 1$- approximation ratio
C - the maximum allowed cost of a path

```
1  δ ← (1 + ε/6H)
2  for all v ∈ V do
3    T_(s,v)[0].w ← ∞
4  T_(s,s)[0].w ← 0, T_(s,s)[0].p ← nil
5  i_max ← min_{i=1,2,···} {i | C ≤ δ^i}
6  for i ← 1 to i_max do
7    c_i ← δ^i
8    for all v ∈ V do
9      T_(s,v)[c_i].w ← T_(s,v)[c_{i-1}].w
10     T_(s,v)[c_i].p ← T_(s,v)[c_{i-1}].p
11   for each link e(v,u) ∈ E do
12     if(c_e ≤ c_i) then
13       c ← max_{j=1,2,···} {δ^j | δ^j ≤ c_i − c_e}
14       if T_(s,v)[c].w + w_e ≤ T_(s,u)[c_i].w then
15         T_(s,u)[c_i].w ← T_(s,v)[c].w + w_e
16         T_(s,u)[c_i].p ← v
```

Figure 10: Algorithm RSP-GEN

$i = 27$ we have $T_{(s,t)}[c_i].w \leq 15$, where $c_i = \delta^i = 13.11$. We conclude that the algorithm identifies a path whose weight is at most $\hat{w} = 15$, and whose cost is at most $13.11 \leq (1+\varepsilon)C(\hat{\mathcal{P}}) = 22$. In fact, Algorithm RSP-GEN applied for G, s, $\varepsilon = 2$ and $\hat{w} = 15$, yields a path $\{s, u_1, u_2, t\}$ whose cost is 13, which is 1.18 times more than the optimum (11).

Algorithm RSP-GEN constitutes the first phase of our precomputation scheme, and its output, *i.e.*, the arrays $T_{(s,v)}[c]$, is used by the second phase. That phase is invoked upon a connection request between s and a destination node $t \in V$, with a QoS requirement $\hat{w}$.

Upon arrival of a connection request for an (s,t)-path $\hat{\mathcal{P}}$ with a QoS requirement $\hat{w}$, we first find the cost $\hat{c}$ of $\hat{\mathcal{P}}$ by setting $\hat{c} = \min\{c_i \mid T_{(s,t)}[c_i].w \leq \hat{w}\}$, where $c_i = \delta^i$. This operation is performed through a binary search on $\mathcal{O}(\frac{1}{\varepsilon} H \log C)$ values of c_i, and requires $\mathcal{O}(\log(\frac{1}{\varepsilon} H \log C))$ time. The running time can be improved by considering only $\mathcal{O}(\frac{1}{\varepsilon}\log C)$ values of c_i, namely $c_i = \min\{\delta^j \mid \delta_1^i \leq \delta^j\}$, where $\delta_1 = (1 + \varepsilon/3)$. This improvement yields a running time of $\mathcal{O}(\log(\frac{1}{\varepsilon}\log C))$, and, as we prove below, does not introduces a penalty in terms of approximation's accuracy.

Next, we identify a suitable path $\hat{\mathcal{P}} = \{s = v_0, \cdots, v_h = t\}$ by using the information stored in the arrays $\{T_{(s,v)}[c] \mid v \in G\}$. Specifically, the predecessor v_{h-1} of t in $\hat{\mathcal{P}}$ is determined by setting $v_h = T_{(s,t)}[\hat{c}].p$. Generally, the predecessor v_{i-1} of v_i is determined by setting $v_{i-1} = T_{(s,v_i)}[x_i].p$, where $x_i = \max_{j=1,2,\cdots,i_{max}} \{\delta^j \mid \delta^j \leq \hat{c} - C(\hat{\mathcal{P}}_{(v_i,t)})\}$ and $C(\hat{\mathcal{P}}_{(v_i,t)})$ is the cost of the subpath $\hat{\mathcal{P}}_{(v_i,t)}$ of $\hat{\mathcal{P}}$ discovered so far.

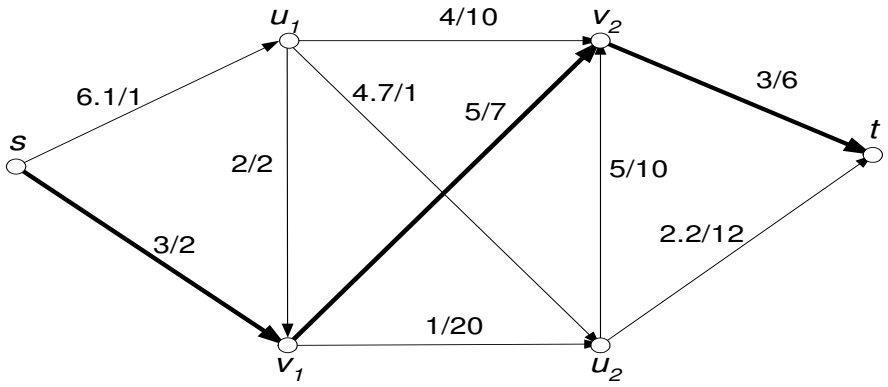


Figure 11: Execution of Algorithm RSP-GEN. For each link e the upper number shows c_e and the lower number shows w_e.

5.2.1 Analysis of the Precomputation Scheme

Lemma 10 *Given are a graph G, a source node s and an approximation parameter ε. For a (arbitrary) value $\hat{w}$ and a (arbitrary) destination node $t \in G$, let c^{opt} be the cost of a $\hat{w}$-weight constrained optimal (s,t)-path, and let $\hat{\mathcal{P}}$ be the path identified by using the arrays $\{T_{(s,v)}[c]\}$, as described above. Then, $\frac{C(\hat{\mathcal{P}})-c^{opt}}{c^{opt}} \leq \varepsilon$.*

Proof: See Appendix E. ■

Lemma 11 *The computational complexity of Algorithm* RSP-GEN *is $\mathcal{O}(\frac{1}{\varepsilon}MH\log C)$.*

Proof: Let us count the number of iterations i_{max} of the algorithm's main loop (*i.e.*, the loop beginning on line 6). Clearly, $\delta^{i_{max}-1} \leq C$, thus $i_{max} \leq \frac{\log C}{\log(\delta)} + 1$. Since for all $x \geq 0$ it holds that $\log(1+x) \geq \frac{x}{(1+x)}$, we have that $\log(\delta) = \log(1+\frac{\varepsilon}{6H}) \geq \frac{\varepsilon}{6H+\varepsilon}$. Thus, $i_{max} \leq \frac{(6H+\varepsilon)\log C}{\varepsilon} + 1 = \mathcal{O}(\frac{H\log C}{\varepsilon})$. Each iteration of the main loop requires $\mathcal{O}(M)$ time, hence the complexity of the algorithm is $\mathcal{O}(\frac{1}{\varepsilon}MH\log C)$. ■

The above results are summarized in the following theorem.

Theorem 4 *Algorithm* RSP-GEN *computes, in $\mathcal{O}(\frac{1}{\varepsilon}MH\log C)$ time, an ε-approximate solution of Problem ACOP.*

5.3 Discussion

We established a precomputation scheme for Problem RSP that provides ε-optimal solutions within a computational complexity of $\mathcal{O}(\frac{1}{\varepsilon}HM\log C)$ for the first phase and $\mathcal{O}(\log(\frac{1}{\varepsilon})+\log\log C)$ for the second phase.

Compared to an alternative single-phase (*i.e.*, "no precomputation") scheme, our scheme allows to (significantly) reduce the time required for establishing a new connection. Indeed, in a single-phase scheme, Problem RSP should be solved for each connection request, through an ε-optimal approximation to Problem RSP [26], which incurs a computational complexity of $\mathcal{O}(MH(\frac{1}{\varepsilon}+\log\log H))$. We conclude that the second phase of our scheme allows to identify an ε-optimal path upon a connection request $\Omega(\frac{MH}{\varepsilon(\log(1/\varepsilon)+\log\log C)})$ times faster.

As previously noted, a precomputation scheme can be trivially constructed on the basis of existing approximation algorithms for Problem RSP, such as [26], by sequentially executing them for various weight values. In order to perform the precomputation for Problem RSP, this algorithm should be invoked $\mathcal{O}\left(\frac{1}{\varepsilon}\log C\right)$ times *per destination*, with a total complexity of $\mathcal{O}(\frac{1}{\varepsilon^2}NHM\log C)$ for all destinations, which is significantly ($\Omega(N/\varepsilon)$ times) higher than that of our solution.

6 Conclusion

QoS routing poses major challenges in terms of algorithmic design. On one hand, the path selection process is a complex task, due to the need to concurrently deal with the connection's QoS requirements, as well as with the global utilization of network resources; on the other hand, connection requests need to be handled promptly upon their arrival, hence there is limited time to spend on path selection. In many practical cases, a precomputation scheme offers a suitable solution to the problem: a background process (the "first phase") prepares a database, which enables to identify a suitable path upon each connection request, through a simple, fast, procedure (the "second phase").

While much work has been done in terms of path selection algorithms for QoS routing, the precomputation perspective received little attention. As was demonstrated in this study, simplistic adaptations of standard algorithms are usually inefficient.

Accordingly, this study investigated the precomputation perspective, considering two major settings of QoS routing. First, we focused on the (practically important) special case where the QoS constraint is of the "bottleneck" type, *e.g.*, a bandwidth requirement, and network optimization is sought through hop minimization. For this setting, the standard Bellman-Ford algorithm offers a straightforward precomputation scheme. However, we showed that, by exploiting the typical hierarchical structure of large-scale networks, one can achieve a substantial improvement in terms of computational complexity.

Next, we considered networks with topology aggregation, which is an inevitable tool for providing scalable routing. We indicated that precomputation is an inherent component of QoS routing schemes in aggregated environments. Accordingly, we extended our precomputation scheme for bottleneck QoS requirements, in a way that is suitable for topology aggregation. This specific extension indicates how our precomputation techniques can be adapted to aggregated environments in general.

Then, we turned to consider the second setting, namely "additive" QoS constraints (*i.e.*, delay) and general link costs. As the related routing problem is $\mathcal{NP}$-hard, we focused on ε-optimal approximations, and derived a precomputation scheme that offers a major improvement over the "standard" approach.

Finally, we note that the precomputation concept is applicable to various areas of network control and management, hence offering a rich ground for future research.

Appendix

A Detailed Description of Procedure PATH

We begin by presenting Procedure GET-PATH that retrieves the paths (pre)computed by Procedure CLUSTER. Next, we present Procedure PATH that identifies the required QoS path by concatenating paths returned by Procedure GET-PATH.

A.1 Procedure GET-PATH

Procedure GET-PATH receives as input a layer-k peer group Γ, a pair of Γ's border nodes b_i, b_j and a hop constraint $\hat{h}$. The procedure uses the output of Procedure CLUSTER to identify a (b_i, b_j)-path $\hat{\mathcal{P}}$ such that $|\hat{\mathcal{P}}| \leq \hat{h}$ and $W(\hat{\mathcal{P}}) = F^{\Gamma}_{(b_i,b_j)}(\hat{h})$.

Procedure GET-PATH ($\Gamma, b_i, b_j, \hat{h}$)

input:

Γ- layer-k peer group

b_i, b_j - Γ's border nodes

$\hat{h}$- hop count constraint

1 $v \leftarrow b_j$, $\hat{\mathcal{P}} \leftarrow \emptyset$
2 **while** $\hat{h} - |\hat{\mathcal{P}}| > 0$ **do**
3 $u = T^{\Gamma}_{(b_i,v)}[\hat{h} - |\hat{\mathcal{P}}|].p$
4 $h = T^{\Gamma}_{(b_i,v)}[\hat{h} - |\hat{\mathcal{P}}|].c$
5 **if** u, v are border nodes of a children Γ' of Γ **then**
6 $\hat{\mathcal{P}}' \leftarrow$GET-PATH$(\Gamma', u, v, h)$
7 $\hat{\mathcal{P}} \leftarrow \hat{\mathcal{P}}' \circ \hat{\mathcal{P}}$
8 **else**
9 $\hat{\mathcal{P}} \leftarrow \{u, v\} \circ \hat{\mathcal{P}}$
10 $v \leftarrow u$
11 **return** $\hat{\mathcal{P}}$

Figure 12: Procedure GET-PATH

If Γ is a layer-1 peer group, then we use the following procedure. We first discover the predecessor v_1 of b_j, then the predecessor v_2 of v_1, *etc*. The predecessor v_{l+1} of v_l is determined by setting $v_{l+1} = T^{\Gamma}_{(b_i,v_l)}[\hat{h} - l].p$. The procedure returns the path $\hat{\mathcal{P}} = \{b_i = v_{\hat{h}}, \cdots, v_1, b_j\}$.

If Γ is a layer-k peer group, then we need a more elaborated procedure, because $\hat{\mathcal{P}}$ runs through children of Γ. We first identify the children through which the path $\hat{\mathcal{P}}$ runs. Next, we recursively determine the detailed path through each (layer-$(k-1)$) child crossed by $\hat{\mathcal{P}}$. Specifically, beginning with b_j, we iteratively discover the predecessor u of each node v, such that $u \in \bar{G}$. This is done by setting $u = T^{\Gamma}_{(b_i,v)}[\hat{h} - C(\hat{\mathcal{P}}_{(v,b_j)})].p$, where $\hat{\mathcal{P}}_{(v,b_j)}$ is the subpath of $\hat{\mathcal{P}}$ identified so far. If u and v are border nodes of some child Γ' of G, then the subpath $\hat{\mathcal{P}}_{(u,v)}$ of $\hat{\mathcal{P}}$ is determined by invoking Procedure GET-PATH on Γ'. Otherwise, $\hat{\mathcal{P}}_{(u,v)}$ comprises of the link $(u, v) \in \Gamma$. We continue this process till we reach b_i.

The formal specification of Procedure GET-PATH appears in Fig. 12.

A.2 Procedure PATH

Procedure PATH that identifies, for a peer group Γ^k and a border node $b_i \in \Gamma^k$, a $\hat{w}$-weight constrained optimal (b_i, t)-path. If $k = 1$, *i.e.*, Γ^1 is a layer-1 peer group, such a path was identified by the BFS algorithm. For $k > 1$, Procedure FIND yields the minimum cost (b_i, t)-path $\bar{\mathcal{P}}^{\Gamma^k}(b_i, t)$ in the auxiliary graph $\bar{\Gamma^k}$. Procedure PATH identifies a path in Γ^k that corresponds to $\bar{\mathcal{P}}^{\Gamma^k}(b_i, t)$. For each link $e \in \bar{\mathcal{P}}^{\Gamma^k}(b_i, t)$, one of the following cases applies:

1. e connects border nodes b_{ij} and $b_{i'j'}$ of different children of Γ^k. In this case, link e is substituted by a link $(b_{ij}, b_{i'j'}) \in \Gamma^k$.

2. e connects border nodes b_{ij} and $b_{ij'}$ of the same child Γ_i of Γ^k. In this case, we substitute e with the $(b_{ij}, b_{ij'})$-path across Γ_i, which is identified by Procedure GET-PATH (Section A.1).

3. e connects a border node b_j of Γ^{k-1} and the destination t. In this case, Procedure PATH is applied

Procedure PATH ($\Gamma^k, \bar{\mathcal{P}}$)

input:

Γ^k - layer-k peer group

$\bar{\mathcal{P}}$- the path computed by Procedure FIND

1 **if** $k = 1$ **then**
2 **return** the path computed in Line 3
3 $\hat{\mathcal{P}} \leftarrow \emptyset$
4 **for** each link $e(u_l, u_{l+1}) \in \bar{\mathcal{P}}$ in order of it appearance in the path **do**
5 **if** u_l, u_{l+1} are border nodes of Γ^{k-1} **then**
6 let $\bar{\mathcal{P}}'$ be the path determined in Line 19 for Γ^{k-1}
7 $\hat{\mathcal{P}}' \leftarrow$PATH$(\bar{\mathcal{P}}', \Gamma^{k-1})$
8 $\hat{\mathcal{P}} \leftarrow \hat{\mathcal{P}} \circ \hat{\mathcal{P}}'$
9 **else if** u_l and u_{l+1} are border nodes of a child Γ' of Γ **then**
10 $\hat{\mathcal{P}}' \leftarrow$GET-PATH$(\Gamma', u_l, u_{l+1}, c_e)$
11 $\hat{\mathcal{P}} \leftarrow \hat{\mathcal{P}} \circ \hat{\mathcal{P}}'$
12 **else**
13 $\hat{\mathcal{P}} \leftarrow \hat{\mathcal{P}} \circ (u_l, u_{l+1})$
14 **return** $\hat{\mathcal{P}}$

Figure 13: Procedure PATH

recursively for Γ^{k-1} and border node $b_j \in \Gamma^{k-1}$. The link e is then substituted by a path returned by the recursive invocation of Procedure PATH.

The formal specification of Procedure PATH appears in Fig. 13.

B Proof of Lemma 3

Lemma 3 Suppose that the (correct) traversal function $F^{\Gamma_i}_{(b_{ij},b_{ij'})}(h)$ is available for each pair $(b_{ij}, b_{ij'})$ of border nodes of each child Γ_i of Γ. Then, Procedure CLUSTER, applied on Γ and a border node b_0 of Γ, computes the traversal function $F^{\Gamma}_{(b_0,b_i)}(h)$ for each border node b_i of Γ.

Proof: Through Procedure GET-PATH we can identify, for each border node b_i of Γ and each $0 \leq h \leq H_\Gamma$, a path $\hat{\mathcal{P}} \in \Gamma$ such that $|\hat{\mathcal{P}}| \leq h$ and $W(\hat{\mathcal{P}}) = F^{\Gamma}_{(b_0,b_i)}[h]$.

Next, we prove that, for each (b_0, v)-path $\bar{\mathcal{P}}$ in $\bar{\Gamma}$, it holds that $T^{\Gamma}_{(b_0,v)}[C(\bar{\mathcal{P}})].w \leq W(\bar{\mathcal{P}})$. By way of contradiction, assume that there exist paths for which this condition does not hold. Let $\bar{\mathcal{P}} = \{b_0, \cdots, v\}$ be such a path of minimum hop count. We denote by u the last predecessor of v in $\bar{\mathcal{P}}$ and by e the link (v, u) of $\bar{\mathcal{P}}$. Since $|\bar{\mathcal{P}}_{(b_0,u)}| < |\bar{\mathcal{P}}|$ it holds that $T^{\Gamma}_{(b_0,u)}[C(\bar{\mathcal{P}}_{(b_0,u)})].w \leq W(\bar{\mathcal{P}}_{(b_0,u)})$. We consider two possible cases.

1. When procedure PROPAGATE is invoked with input e in line 12, it holds that $T^{\Gamma}_{(b_0,u)}[C(\bar{\mathcal{P}}_{(b_0,u)})].w \leq W(\bar{\mathcal{P}}_{(b_0,u)})$. In this case, after the execution of the procedure, $T^{\Gamma}_{(b_0,v)}[C(\bar{\mathcal{P}})].w \leq W(\bar{\mathcal{P}})$, hence resulting in a contradiction.

2. Otherwise, consider the step of the procedure in which $T^{\Gamma}_{(b_0,u)}[C(\bar{\mathcal{P}}_{(b_0,u)})].w$ was assigned the value $w \leq W(\bar{\mathcal{P}}_{(b_0,u)})$. Since link e was already processed by the loop at line 8, this update leads to a recursive invocation of the procedure PROPAGATE (line 6) for a link $e'(u, v)$, where $c_{e'} \leq c_e$, and, again, after this invocation, $T^{\Gamma}_{(b_0,v)}[C(\bar{\mathcal{P}})].w \leq W(\bar{\mathcal{P}})$, resulting in a contradiction.

Let $\hat{\mathcal{P}}$ be a (b_0, b_i)-path in Γ. By Lemma 2 there exists a (b_0, b_i)-path $\bar{\mathcal{P}}$ in $\bar{\Gamma}$ such that $C(\bar{\mathcal{P}}) \leq |\hat{\mathcal{P}}|$ and $W(\bar{\mathcal{P}}) \leq W(\hat{\mathcal{P}})$. Since $T^{\Gamma}_{(b_0,b_i)}[C(\bar{\mathcal{P}})].w \leq W(\bar{\mathcal{P}})$, it holds that $T^{\Gamma}_{(b_0,b_i)}[|\hat{\mathcal{P}}|].w = F^{\Gamma}_{(b_0,b_i)}[|\hat{\mathcal{P}}|] \leq W(\hat{\mathcal{P}})$, which completes the proof of the lemma. ■

C Proof of Lemma 6

Lemma 6 Procedure FIND, invoked for a source s, destination t and QoS constraint $\hat{w}$, returns the minimum hop count of a $\hat{w}$-weight constrained (s, t)-path.

Proof: Let $\hat{k}$ be the lowest hierarchical layer such that there exists a layer-$\hat{k}$ peer group $\Gamma^{\hat{k}}$ for which s and t are border nodes.

We prove the following assertion by induction on k: For each peer group Γ^k, $1 \leq k \leq \hat{k}$, and for each border node b of Γ^k, Procedure FIND determines the minimum hop count $H_{(b_i,t)}$ of a $\hat{w}$-weight constrained (b_i, t) path in Γ^k.

As a base step, consider the parent Γ^1 of t. After pruning from Γ all links e such that $w_e \geq \hat{w}$, each path in the resulting graph satisfies the weight constraint. Thus, the assertion follows from the correctness of the BFS algorithm.

Assume inductively that the procedure determines, for each border node b_j of Γ^{k-1}, the correct value of $H_{(b_j,t)}$. Let b_i be a border node of Γ^k and let $\hat{\mathcal{P}}$ be the minimum hop count of a $\hat{w}$-weight constrained (b_i, t)-path. Let $\bar{\mathcal{P}}$ be a (b_i, t)-path in $\bar{\Gamma}$ that corresponds to $\hat{\mathcal{P}}$, *i.e.*, $\bar{\mathcal{P}} = \{b_i = u_0, u_1, \cdots, t\}$, where $u_0 = b_i$ and u_l is a first node in $\hat{\mathcal{P}}$ after u_{l-1} that belongs to $\bar{\Gamma}$. We show that $C(\bar{\mathcal{P}}) \leq |\hat{\mathcal{P}}|$. Let (u_{l-1}, u_l) be a link in $\bar{\mathcal{P}}$. We consider several cases:

1. (u_{l-1}, u_l) connects two border nodes of some child Γ of Γ^k. Then, by Theorem 1, $F^{\Gamma}_{(u_{l-1},u_l)}(|\hat{\mathcal{P}}_{(u_{l-1},u_l)}|) \leq W(\hat{\mathcal{P}}_{(u_{l-1},u_l)})$. Hence the cost of the link (u_{l-1}, u_l) is at most $|\hat{\mathcal{P}}_{(u_{l-1},u_l)}|$.

2. (u_{l-1}, u_l) connects two border nodes of different children of Γ^k. Then, $(u_{l-1}, u_l) \in \hat{\mathcal{P}}$. Since the cost of link (u_{l-1}, u_l) is assigned a unit cost, $c_{(u_{l-1},u_l)} \leq |\hat{\mathcal{P}}_{(u_{l-1},u_l)}|$.

3. (u_{l-1}, u_l) connects a border node of Γ^{k-1} and the destination t, *i.e.*, $u_l = t$. Then, the inductive argument implies that $c_{(u_{l-1},u_l)} = H_{(u_{l-1},u_l)} \leq |\hat{\mathcal{P}}_{(u_{l-1},u_l)}|$.

The above case analysis implies that, for each link (u_{l-1}, u_l) in $\bar{\mathcal{P}}$, it holds that $c_{(u_{l-1},u_l)} \leq |\hat{\mathcal{P}}_{(u_{l-1},u_l)}|$. Hence, $C(\bar{\mathcal{P}}) \leq |\hat{\mathcal{P}}|$, and Dijksta's algorithm returns a path $\bar{\mathcal{P}}'$ whose cost is at most $|\hat{\mathcal{P}}|$. Let $\hat{\mathcal{P}}'$ be a path in the actual network that corresponds to $\bar{\mathcal{P}}'$. Since $|\hat{\mathcal{P}}'| \leq C(\bar{\mathcal{P}}') \leq |\hat{\mathcal{P}}|$ and $W(|\hat{\mathcal{P}}'|) \leq \hat{w}$, we conclude that $\hat{\mathcal{P}}'$ is a minimum hop $\hat{w}$-weight constrained path, which implies the correctness of the assertion and the lemma. ■

D Proof of Lemma 8

Lemma 8 Algorithm AGG-AHOP computes the aggregate traversal function $F^{\hat{G}}_{(s,t)}(c)$ for each node t of $\hat{G}$.

Proof: Let $\hat{w}$ be a (arbitrary) QoS constraint and v be a (arbitrary) node in $\hat{G}$.

First, we note that there exists an (s, v)-path $\hat{\mathcal{P}}$, such that $T^{\hat{G}}_{(s,v)}[C(\hat{\mathcal{P}}, \hat{w})].w = \hat{w}$. Path $\hat{\mathcal{P}}$ can be identified by using arrays $T^{\hat{G}}_{(s,v)}$, as described in Section 4.3.

Next, we prove that, for each (s,v)-path $\hat{\mathcal{P}}$ in $\hat{G}$, $T^{\hat{G}}_{(s,v)}[C(\hat{\mathcal{P}},\hat{w})].w \leq \hat{w}$. By way of contradiction, assume that there exist paths for which this condition does not hold. Let $\hat{\mathcal{P}} = \{s,\cdots,v\}$ be such a path of minimum cost. We denote by u the last predecessor of v in $\hat{\mathcal{P}}$ and by e the link (u,v) of $\hat{\mathcal{P}}$. Since $C(\hat{\mathcal{P}}_{(s,u)}) < C(\hat{\mathcal{P}})$ it holds that $T^{\hat{G}}_{(s,u)}[C(\hat{\mathcal{P}}_{(s,u)},\hat{w})].w \leq \hat{w})$. We consider two possible cases.

1. When procedure PROPAGATE is invoked with input e in line 12 of the algorithm, it holds that $T^{\hat{G}}_{s,u)}[C(\hat{\mathcal{P}}_{(s,u)},\hat{w})].w \leq \hat{w})$. In this case, after the execution of the procedure, $T^{\hat{G}}_{(s,v)}[C(\hat{\mathcal{P}},\hat{w})].w \leq \hat{w}$, hence resulting in a contradiction.

2. Otherwise, consider the step of the algorithm in which $T^{\hat{G}}_{(s,u)}[C(\hat{\mathcal{P}}_{(s,u)},\hat{w})].w$ was assigned the value $\hat{\hat{w}} \leq \hat{w})$. Since link e was already processed by the loop at line 8, this update leads to a recursive invocation of the procedure PROPAGATE (line 6) for a link $e'(u,v)$, where $c_{e'} \leq c_e$, and, again, after this invocation, $T^{\hat{G}}_{(s,v)}[C(\hat{\mathcal{P}},\hat{w})].w \leq \hat{w}$, resulting in a contradiction.

Since $\hat{w}$ and v are arbitrary, the lemma follows. ■

E Proof of Lemma 10

Lemma 10 Given are a graph G, a source node s and an approximation parameter ε. For a (arbitrary) value $\hat{w}$ and a (arbitrary) destination node $t \in G$, let c^{opt} be the cost of a $\hat{w}$-weight constrained optimal (s,t)-path, and let $\hat{\mathcal{P}}$ be the path identified by using the arrays $\{T_{(s,v)}\}$, as described above. Then, $\frac{C(\hat{\mathcal{P}})-c^{opt}}{c^{opt}} \leq \varepsilon$.

Proof: Let $\mathcal{P}^{opt} = \{v_0 = s, v_1, ..., v_h = t\}$ be a $\hat{w}$-weight constrained optimal (s,t)-path. We denote by $\hat{c}_{v_j} = \min_{i=1,2,...,i_{max}} \left\{\delta^i | T_{(s,v_j)}[\delta^i].w \leq W(\mathcal{P}^{opt}_{(s,v_j)})\right\}$.

We prove that, for $1 \leq j \leq h$, $\hat{c}_{v_j} \leq \delta^j C(\mathcal{P}^{opt}_{(s,v_j)})$. The base step is straightforward, since $C(\mathcal{P}^{opt}_{(s,s)}) = \hat{c}_s = 0$. For the inductive step, we assume that $\hat{c}_{v_j} \leq \delta^j C(\mathcal{P}^{opt}_{(s,v_j)})$ holds for $1,2,..,j-1$ and prove that it holds for j. Let us consider the execution of the the main loop, *i.e.*, the loop at line 6, for $i = \min\{l | \delta^l \geq \hat{c}_{v_{j-1}} + c_{(v_{j-1},v_j)}\}$. Since $\hat{c}_{v_{j-1}} < c_i$, the value of $T_{(s,v_{j-1})}[\hat{c}_{v_{j-1}}].w$ was set at a previous iteration of the main loop. As a result, and since $T_{(s,v_{j-1})}[\hat{c}_{v_{j-1}}].w \leq W(\mathcal{P}^{opt}_{(s,v_{j-1})})$, lines 14 and 15 assure that $T_{(s,v_j)}[c_i].w \leq W(\mathcal{P}^{opt}_{(s,v_j)})$. Therefore, $\hat{c}_{v_j} \leq c_i \leq \delta \cdot (\hat{c}_{v_{j-1}} + c_e) \leq \delta \cdot (C(\mathcal{P}^{opt}_{(s,v_{j-1})}) \cdot \delta^{j-1} + c_e)$, for $e = (v_{j-1}, v_j)$, where the last inequality follows from the inductive assumption. Since $C(\mathcal{P}^{opt}_{(s,v_j)}) = C(\mathcal{P}^{opt}_{(s,v_{j-1})}) + c_e$, we have $\hat{c}_{v_j} \leq C(\mathcal{P}^{opt}_{(s,v_j)}) \cdot \delta^j$.

We proved that $\hat{c}_t = C(\hat{\mathcal{P}}) \leq \delta^h C(\mathcal{P}^{opt})$. From the relation $\log(1+x) \leq x$ for $x \geq 1$, it follows that, for $xh \leq 1$, $(1+x)^h \leq \frac{1}{1-xh}$. Thus, $\delta^h = (1+\frac{\varepsilon}{6H})^h \leq \frac{1}{1-\varepsilon h/(6H)} \leq \frac{1}{1-\varepsilon/6} = 1 + \frac{\varepsilon}{3(2-\varepsilon/3)} \leq 1+\varepsilon/3$, where the last inequality follows from the fact that $\varepsilon \leq 1$. Thus, $\hat{c}_t \leq (1+\varepsilon/3)C(\mathcal{P}^{opt})$. Since $\delta_1 = 1+\varepsilon/3$, we have $C(\hat{\mathcal{P}}) \leq (1+\varepsilon/3)\hat{c}_t$. We conclude that $C(\hat{\mathcal{P}}) \leq (1+\varepsilon/3)^2 C(\mathcal{P}^{opt}) \leq (1+\varepsilon)C(\mathcal{P}^{opt})$. Hence, $\frac{C(\hat{\mathcal{P}})-C(\mathcal{P}^{opt})}{C(\mathcal{P}^{opt})} \leq \varepsilon$, and the lemma follows. ■

References

[1] G. Apostolopoulos, R. Guérin, S. Kamat, A. Orda, T. Przygienda, and D. Williams. QoS Routing Mechanisms and OSPF Extensions. RFC No. 2676. Internet Engineering Task Force, August 1999.

[2] G. Apostolopoulos, R. Guérin, S. Kamat, A. Orda, and S. K. Tripathi. Intra-Domain QoS Routing in IP Networks: A Feasibility and Cost/Benefit Analysis. *IEEE Network (Special Issue on Integrated and Differentiated Services for the Internet)*, 13(5):42–54, September-October 1999.

[3] G. Apostolopoulos, R. Guérin, S. Kamat, and S. Tripathi. Quality of Service Based Routing: A Performance Perspective. In *Proceedings of SIGCOMM*, pages 17–28, Vancouver, Ontario, Canada, September 1998.

[4] B. Awerbuch and Y. Shavitt. Topology Aggregation for Directed Graphs. *IEEE/ACM Transactions on Networking*, 9(1):82–90, February 2001.

[5] D. Bauer, J.N. Daigle, I. Iliadis, and P. Scotton. Efficient Frontier Formulation for Additive and Restrictive Metrics in Hierarchical Routing. In *Proceedings of IEEE ICC 2000*, New Orleans, LA, USA, June 2000.

[6] A. Bestavros and I. Matta. Load Profiling for Efficient Route Selection in Multi-Class Networks. In *Proceedings of IEEE ICNP'97*, Atlanta, GA, USA, October 1997.

[7] J.-Y. Le Boudec and T. Przygienda. A Route Pre-Computation Algorithm for Integrated Services Networks. *Journal of Network and Systems Management*, 3(4):427–449, December 1995.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[9] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick. A Framework for QoS-based Routing in the Internet. RFC No. 2386. Internet Engineering Task Force, August 1998.

[10] G. Frederickson and D. Johnson. The Complexity of Selection and Ranking in X + Y and Matrices with Sorted Columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, CA, USA, 1979.

[12] R. Guérin and A. Orda. QoS-based Routing in Networks with Inaccurate State and Metrics Information: Theory and algorithms. *IEEE/ACM Transactions on Networking*, 7(3):350–364, June 1999.

[13] R. Guérin and A. Orda. Computing Shortest Paths for any Number of Hops. *IEEE/ACM Transactions on Networking*, 10(5):613–620, October 2002.

[14] R. Hassin. Approximation Schemes for the Restricted Shortest Path Problem. *Mathematics of Operations Research*, 17(1):36–42, February 1992.

[15] T. Korkmas and M. Krunz. Source-Oriented Topology Aggregation with Multiple QoS Parameters in Hierarchical Networks. *ACM Transactions on Modeling and Computer Simulation*, 10(4):295–325, October 2000.

[16] W. C. Lee. Topology Aggregation for Hierarchical Routing in ATM Networks. In *Proceedings of ACM SIGCOMM'95*, Cambridge, MA, USA, April 1995.

[17] D. H. Lorenz and A. Orda. QoS Routing in Networks with Uncertain Parameters. *IEEE/ACM Transactions on Networking*, 6(6):768–778, December 1998.

[18] D. H. Lorenz and A. Orda. Optimal Partition of QoS Requirements on Unicast Paths and Multicast Trees. *IEEE/ACM Transactions on Networking*, 10(1):102–114, February 2002.

[19] K.S. Lui and K. Nahrstedt. Topology Aggregation and Routing in Bandwidth-Delay Sensitive Networks. In *Proceedings of IEEE Globecom'2000*, San Francisco, CA, USA, November-December 2000.

[20] Q. Ma and P. Steenkiste. Quality of Service Routing for Traffic with Performance Guarantees. In *Proceedings of IWQoS'97*, Columbia University, New York, NY, May 1997.

[21] J. Moy. OSPF Version 2. RFC No. 2328. Internet Engineering Task Force, April 1998.

[22] A. Orda. Routing with End to End QoS Guarantees in Broadband Networks. *IEEE/ACM Transactions on Networking*, 7(3):365–374, June 1999.

[23] D. Peleg and A.A. Shcaffer. Graph Spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[24] C. Pornavalai, G. Chakraborty, and N. Shiratori. QoS Based Routing Algorithm in Integrated Services Packet Networks. In *Proceedings of IEEE ICNP'97*, Atlanta, GA, USA, October 1997.

[25] Private Network-Network Interface Specification v1.0 (PNNI). ATM Forum Technical Committee, March 1996.

[26] D. Raz and D. H. Lorenz. A Simple Efficient Approximation Scheme for the Restricted Shortest Path Problem. *Operations Research Letters*, 28(5):213–219, June 2001.

[27] A. Shaikh, J. Rexford, and K. Shin. Efficient Precomputation of Quality-of-Service Routes. In *Proceedings of Workshop on Network and Operating Systems Support for Audio and Video (NOSSDAV'98)*, Cambride, UK, June 1998.

[28] Z. Wang and J. Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1288–1234, September 1996.