Q-BATE: A QoS Constraint-based Traffic Engineering Routing Algorithm

Stefano Avallone and Giorgio Ventre
COMICS Lab, Dipartimento di Informatica e Sistemistica
Università di Napoli Federico II
Via Claudio 21, 80125 Napoli, Italy
Email: {stavallo, giorgio}@unina.it

Abstract—The problem of finding multi-constrained paths has been addressed by several QoS routing algorithms. While they generally satisfy the application requirements, they often do not consider the perspective of service providers. Service providers aim at optimizing resource usage and maximizing the throughput and the number of accepted requests. These goals have been addressed by traffic engineering algorithms, which usually consider bandwidth as the sole application requirement. We propose a new routing algorithm, Q-BATE, which attempts to optimize network utilization while still offering QoS guarantees. The basic concepts of Q-BATE are look-ahead, depth-first approach and a path length definition as a function of both the available bandwidth and other additive QoS measures. This paper presents a comparative analysis of several proposed algorithms via simulation studies. The simulations show that Q-BATE performs better than the other algorithms and it has a fast running time.

I. INTRODUCTION

Many algorithms [1] have been proposed to find the shortest path subject to multiple additive QoS constraints. This problem, called MCOP (Multi-Constrained Optimal Path), is NP-complete. Therefore, mainly heuristics have been proposed to find an approximate solution to the MCOP problem. As such, they attempt to achieve a solution close to the optimum, while keeping computational complexity low. Their performance is usually evaluated by assuming a temporarily frozen view of the network and computing a path for one flow request. Since the available bandwidth (a bottleneck QoS measure) is not considered, they are not analyzed under an important point of view: the evolution of network resources utilization as several flows are routed. Such kind of analysis enables to compare the effectiveness of routing algorithms in a dynamic scenario. Indeed, different algorithms select different paths for flow requests, leading to a different occupation of resources. The consequence is that the algorithms differ in their ability of admitting flow requests and maximizing network throughput.

The analysis of routing algorithms presented in this work is focused on resource optimization in a dynamic scenario. Many algorithms, denoted as traffic engineering algorithms, have been proposed which embrace such viewpoint. While focused on the behaviour in a dynamic scenario, most of them do not take into account additive QoS constraints and only consider bandwidth.

In this paper we present a new routing algorithm, Q-BATE (QoS constraint-BAsed Traffic Engineering), that aims at maximizing throughput (or minimizing blocking), while satisfying the users’ QoS requirements. It is our goal to combine these two objectives as efficiently as possible. Q-BATE finds the path with the smallest length among those obeying the constraints. In order to optimize the usage of network resources, we defined the path length as a function of both the available bandwidth and other additive QoS measures. We developed a new routing algorithm tailored to the proposed path length function and based on the look-ahead concept and a depth-first strategy.

The performance of Q-BATE and other algorithms is evaluated through simulations. All the considered algorithms process flow requests one-by-one and do not make use of any a-priori knowledge about either predicted traffic or future demands. We assume the knowledge of quasi-static information such as the network topology and the set of ingress-egress nodes of the network. The only dynamic information is the residual bandwidth (i.e. the portion of the link capacity not yet reserved) of each link in the network.

This paper is structured as follows. In Section II we give a formal definition of the considered routing problem. Some previous QoS routing and traffic engineering algorithms are described in Section III. Section IV introduces the proposed path length function. Section V discusses the two basic concepts of Q-BATE: the look-ahead phase and the depth-first approach. Also, the pseudo-code of our algorithm is presented. Section VI shows a comparative analysis of different routing algorithms. Finally, Section VII concludes our work.

II. PROBLEM STATEMENT

The network is modelled as a graph $G(N, E)$, where $N$ is the set of nodes and $E$ is the set of links. With a slight abuse of notation we will also denote by $N$ and $E$, respectively, the number of nodes and the number of links. Each link $l \in E$ is assigned an $(m + 1)$-dimensional QoS link weight vector $\vec{w}(l) = [w_0(l), w_1(l), \ldots, w_m(l)]$, where $w_0(l)$ is a link weight dependent on the available bandwidth $R(l)$ and the other components are the values of $m$ additive QoS measures. Finally, the capacity of a link $l$ is denoted by $C(l)$.

A flow request is defined by a triple $(s, d, \vec{Q})$, where $s$ is the source node, $d$ is the destination node and $\vec{Q} = [Q_0, Q_1, \ldots, Q_m]$ is a vector representing its QoS requirements. Specifically, $Q_0$ is the requested bandwidth while the other...
components are constraints on the values of the additive QoS measures along the path. Even though minimum (maximum) QoS constraints can be easily dealt with by omitting all links which do not satisfy the requirement, we explicitly consider available bandwidth due to its central role played in resource optimization strategies. Multiplicative QoS measures are not explicitly taken into account because, if we assume independent measures over the links, we can transform them into additive QoS measures by taking the logarithm.

When a flow request arrives, the routing algorithm searches for a feasible path \( P \) that obeys:

\[
\begin{align*}
R(P) & \overset{def}{=} \min_{l \in P} R(l) \geq Q_0 \\
\omega_i(P) & \overset{def}{=} \sum_{l \in P} w_i(l) \leq Q_i, \quad \forall i = 1, \ldots, m
\end{align*}
\]

In case no feasible path is found, the request is rejected. In presence of multiple feasible paths, the algorithm chooses the one which is thought to optimize network utilization. Typically, a path length function is defined and the feasible path with the smallest length is selected.

A. Discussion on additive QoS link weights

This subsection discusses the setting of additive QoS link weights in a dynamic scenario, the guideline being the fulfillment of the QoS requirements of the flows. Indeed, an exact algorithm returns a path \( P \) such that \( \sum_{l \in P} w_i(l) \leq Q_i, i = 1, \ldots, m \). But, the QoS requirements of a flow are satisfied if the experienced QoS is within the constraints, i.e. \( \sum_{l \in P} q_i(l) \leq Q_i, i = 1, \ldots, m \), where \( q_i(l) \) is the value of the \( i \)-th QoS measure experienced crossing link \( l \).

The most intuitive strategy is to set the additive QoS link weights \( (w_i(l)) \) equal to the current experienced values (i.e. \( q_i(l) \)). Hence \( \sum_{l \in P} q_i(l) = \sum_{l \in P} w_i(l) \leq Q_i \) and the QoS constraints are met. However, as a consequence of routing new flows on links of \( P \), the actual QoS values \( q_i(l) \) deteriorate and therefore the QoS granted to already admitted flows may not be preserved. Instead, we propose to set each QoS link weight to a constant value that is independent of the current link status \( q_i(l) \). Such a value must be an upper bound to the actual QoS value, in the sense that if the allocated bandwidth is less than the link capacity, then the QoS values experienced by packets crossing the link do not exceed the QoS weights. This assures that \( \sum_{l \in P} q_i(l) \leq \sum_{l \in P} w_i(l) \leq Q_i, i = 1, \ldots, m \), i.e. the additive QoS constraints will be satisfied even after new flows are routed.

III. RELATED WORK

Van Mieghem and Kuipers proposed SAMCRA (Self Adaptive Multiple Constraints Routing Algorithm) [2], an exact QoS routing algorithm. The path length defined in [2] is a non-linear function of the \( m \) additive QoS measures it considers:

\[
L_S(P) = \max_{1 \leq i \leq m} \frac{w_i(P)}{Q_i}
\]

so that path \( P \) satisfies the constraints when \( L_S(P) \leq 1 \). Like Dijkstra’s algorithm, SAMCRA starts from the source node and visits the neighboring nodes in a breadth-first manner while moving toward the destination node. Unlike Dijkstra, SAMCRA has to store not only the shortest sub-path for each visited node. Indeed, the shortest sub-path from the source to an intermediate node may not lead to any feasible path, while sub-optimal sub-paths may do. In order to reduce complexity (while still returning the exact solution), SAMCRA does not store all the sub-paths but makes an efficient distinction based on non-dominance. A (sub-)path \( P_1 \) is dominated by a (sub-)path \( P_2 \) if \( w_i(P_2) \leq w_i(P_1) \) for \( i = 1, \ldots, m \), with an inequality for at least one link weight component \( i \). SAMCRA only considers non-dominated (sub-)paths.

The path length definition (1) is a function of the QoS link weights \( (w_i(l)) \) and of the QoS constraints \( (Q_i, i = 1, \ldots, m) \). As discussed in Section II-A, static QoS link weights should be considered. Thus, if link weights are load-independent, SAMCRA with such a path length function is expected not to be efficient in a dynamic scenario because network utilization is not taken into account. For this reason we proposed in [3] a new variant, denoted SAMCRA-B, whose path length \( L_{SB} \) is a function of dynamic information such as the available bandwidth. We assumed \( L_{SB}(P) = \sum_{l \in P} w_0(l) \), where \( w_0(l) = \gamma(p_l) \) and \( \gamma(\cdot) \) is an increasing function of the link utilization \( p_l = C(l) - R(l) \) (ratio of the reserved bandwidth to the total capacity). Using a path length function dependent on information other than QoS link weights and QoS constraints requires to reconsider the conditions of the dominance check. In [3] we showed that SAMCRA-B returns the exact solution if we discard a (sub-)path \( P_1 \) when there exists a (sub-)path \( P_2 \) such that \( w_i(P_2) \leq w_i(P_1) \) for \( i = 1, \ldots, m \) and \( L_{SB}(P_2) \leq L_{SB}(P_1) \). The additional condition slightly reduces the efficiency of the search space reduction. But, the simulation in [3] shows that SAMCRA-B outperforms SAMCRA and many traffic engineering algorithms [4]–[7].

There are a few traffic engineering algorithms that explicitly take into account additive QoS constraints. Banerjee and Sidhu [7] proposed two algorithms: TE-B, which takes into account only a bandwidth requirement, and TE-DB, which considers also a delay constraint. The authors introduced three objectives for traffic engineering: (a) reducing the blocking of flows, (b) minimizing network cost and (c) distributing network load. Objective (a) is accomplished analogous to MIRA [8], which uses the decrease in the maxflow [9] between ingress-egress pairs as a measure of the interference due to routing a flow on a certain path. MIRA was an innovative traffic engineering algorithm and inspired several works [5], [6], [10], [11]. The TE-B and TE-DB formulations are proved to be NP-complete [7]. Banerjee and Sidhu presented another formulation in which objective functions (a) and (b) are transformed into constraints. Both TE-B and TE-DB use TAMCRA [12], the predecessor of SAMCRA [2], to find a set of \( k \) paths satisfying the set of constraints and then select the one with the shortest length according to (c).
Hence, both \( \vec{P}_Q \) consume bandwidth along constraint on the second additive QoS measure is not obeyed. Each flow along the feasible path whose weights are the closest to the constraints, flow 1 is routed along \( P_1 \) and does not consume bandwidth along \( P_2 \), which remains feasible for flow 2. Thus, both flows can be accepted.

The routing policy introduced above may be accomplished by properly defining the path length function used by an algorithm that finds a feasible path with the smallest length. Here we propose a path length function made of the product of additive constraints. The first component of the path length (2) decreases as a new link is added to the path, the second component increases. Also, the more the utilization of the link the larger is the increase. Discouraging the use of highly loaded links is a desirable approach because consuming all available bandwidth on some links may cause future requests to be blocked. Thus, the first component drives toward paths whose weights are close to the constraints, while the second promotes short and unloaded paths. The selected path is a compromise between the two objectives. The next section introduces the algorithm we propose to find the feasible path with the shortest length according to (2).

IV. A NEW PATH LENGTH FUNCTION

The idea behind the new path length function proposed in this paper is to select for each flow a path whose QoS weights are smaller than the QoS constraints, but not too much smaller. Otherwise, other flows with stringent requirements might potentially be blocked. We illustrate such a concept with the following example (Figure 1). We suppose \( w_0(l) = R(l) \), \( m = 2 \) and two flows to be routed between \( A \) and \( D \): flow 1 with constraints \( Q_1 = [45 80 120] \) and flow 2 with constraints \( Q_2 = [45 60 80] \). Two paths are possible: path \( P_1 (A \rightarrow B \rightarrow C \rightarrow D) \), having a weight \( \vec{w}(P_1) = [80 50 100] \), and path \( P_2 (A \rightarrow E \rightarrow D) \), having a weight \( \vec{w}(P_2) = [85 10 50] \). Hence, both \( P_1 \) and \( P_2 \) are feasible paths for flow 1. We believe that routing the flows on the path assuring the best QoS is not efficient. Following such strategy, flow 1 in our example would be routed along \( P_2 \). Afterwards, the weight of \( P_2 \) would become \( \vec{w}(P_2) = [40 10 50] \). Then, as both \( P_1 \) (the constraint on the second additive QoS measure is not obeyed) and \( P_2 \) (insufficient available bandwidth) are unfeasible, flow 2 is rejected. If the policy of the routing algorithm is to route each flow along the feasible path whose weights are the closest to the constraints, flow 1 is routed along \( P_1 \) and does not consume bandwidth along \( P_2 \), which remains feasible for flow 2. Thus, both flows can be accepted.

The routing policy introduced above may be accomplished by properly defining the path length function used by an algorithm that finds a feasible path with the smallest length. Here we propose a path length function made of the product of two components, one depending on the additive measures and the other depending on the available bandwidth on the links of the path:

\[
L_Q(P) = f\left(\left|\vec{Q}_a - \vec{w}_a(P)\right|_p\right) \sum_{l \in P} w_0(l) \tag{2}
\]

where \( \vec{w}_a(P) \) is the vector of the additive weights of \( P \) and \( \vec{Q}_a \) is the vector of additive constraints. The first component of \( L_Q(P) \) is a function of the difference vector \( \vec{Q}_a - \vec{w}_a(P) \), while the second one depends on the load of the links constituting the path. We will discuss later on about such functions. Here, we remark that \( f(\cdot) \) must be a non-decreasing function because we want to favour small distances between \( \vec{Q}_a \) and \( \vec{w}_a(P) \) and \( w_0(\cdot) \) must be a positive non-decreasing function of the link utilization because we want to favour slightly loaded links. While the first component of the path length (2) decreases as a new link is added to the path, the second component increases. Also, the more the utilization of the link the larger is the increase. Discouraging the use of highly loaded links is a desirable approach because consuming all available bandwidth on some links may cause future requests to be blocked. Thus, the first component drives toward paths whose weights are close to the constraints, while the second promotes short and unloaded paths. The selected path is a compromise between the two objectives. The next section introduces the algorithm we propose to find the feasible path with the shortest length according to (2).

V. Q-BATE

Q-BATE is a routing algorithm which finds the feasible path having the shortest length according to (2), i.e. the path length we propose to traffic engineer QoS-aware networks. In the following subsections we present the basic concepts and the pseudocode of our algorithm.

A. Look-Ahead

The main concept on which Q-BATE is based is look-ahead. The look-ahead concept has been already used for the purpose of improving the performance by different QoS routing algorithms, e.g. [13], A*Prune [14] and SAMCCA version 2 [2]. It basically consists of a precomputation which aims to make available some information about the remaining sub-path from each node toward the destination. More precisely, the look-ahead phase provides for each node \( u \) in the graph a set of \( m+1 \) paths to the destination \( d \), \( \{S_i(u;v,d)\}_{i=0\ldots m} \), which are the paths that separately minimize each QoS link weight: \( S_i(u;v,d) = \arg\min_{P} w_1(P) \) for \( i = 0, \ldots m \). The information provided by the look-ahead phase can be exploited in a number of ways:

1) Exclusion of neighbors based on additive QoS constraints: When extending a sub-path \( s \sim u \) toward the neighboring nodes of \( u \), it is possible to disregard the neighbors \( v \) for which:

\[
\exists i \in \{1, \ldots m\} \mid w_i(s \sim u) + w_i(u \sim v) + w_i(S_i(u,v,d)) > Q_i \tag{3}
\]

Indeed, \( w_i(S_i(u,v,d)) \) is a lower bound to the \( i \)-th QoS weight of any sub-path from \( v \) to \( d \). If the sum of such lower bound and the \( i \)-th QoS weight of the sub-path \( s \sim u \sim v \) exceeds the \( i \)-th constraint \( Q_i \), then we are guaranteed that the sub-path \( s \sim u \sim v \) cannot be part of a feasible path. We can therefore discard neighbor \( v \), which leads to a reduction in the search space.

2) Initial upper bound to the shortest path length: An outcome of the look-ahead phase is the calculation of \( m+1 \) paths \( \{S_i(s;v,d)\}_{i=0\ldots m} \) between the source \( s \) and the destination \( d \). Some (or even all) of these paths may not satisfy

\[\text{We use the notation } a \sim b \text{ to refer to any path from node } a \text{ to node } b \text{ and } a \sim b \text{ to refer to a specific path } P \text{ from } a \text{ to } b \text{. Instead, the notation } c \sim d \text{ refers to the link between the adjacent nodes } c \text{ and } d.\]
all the constraints and therefore be unfeasible. But, if some of these paths are feasible and we determine the minimum length according to (2), then we obtain an initial upper bound to the length of the shortest feasible path. We denote such upper bound by $\text{maxlength}$. The next point introduces a method to identify and early discard sub-paths that certainly lead to paths having a length larger than $\text{maxlength}$. If the look-ahead phase provides an initial $\text{maxlength}$ value, such a method can be immediately operative.

3) Exclusion of neighbors based on $\text{maxlength}$: In case we found a finite $\text{maxlength}$ value, it is possible to further reduce the search space when extending a sub-path $s \leadsto u$ toward the neighbors of $u$. A neighbor $v$ can be disregarded if:

$$w_0(s \leadsto u) + w_0(u \leadsto v) + w_0(S_{b(v,d)}) > \text{maxlength} \quad (4)$$

provided that $f(x) \geq 1 \forall x \geq 0$. Indeed, for every sub-path $v \leadsto d$ it yields:

$$L_Q(s \leadsto u \leadsto v \leadsto d) \geq w_0(s \leadsto u \leadsto v \leadsto d)$$

$$\geq w_0(s \leadsto u \leadsto v) + w_0(v \leadsto d)$$

$$\geq w_0(s \leadsto u \leadsto v) + w_0(S_{b(v,d)}) > \text{maxlength}$$

The first inequality holds provided that $f(x) \geq 1 \forall x \geq 0$, while the penultimate one derives from the definition of $S_{b(v,d)}$ as the shortest path from $v$ to $d$ according to link weight $w_0(l)$. The previous inequalities show that if (4) is satisfied then it is possible to disregard neighbor $v$ because it certainly leads to paths having a length larger than that of the current shortest path ($\text{maxlength}$).

4) Shortest path search speed-up based on predicted length: Using the information provided by the look-ahead phase it is possible to roughly estimate the length of the shortest path originated by a certain sub-path $s \leadsto u$. Such predicted length, denoted as $E_Q(s \leadsto u)$, is the length of a path composed of the sub-path $s \leadsto u$ and an hypothetic sub-path $u \leadsto d$ whose weights are the minimum weights from $u$ to $d$, i.e. $w_1(u \leadsto d) = w_1(S_{b(u,d)})$ for $i = 0,\ldots,m$:

$$E_Q(s \leadsto u) = f \left( \overrightarrow{Q}_a - \overrightarrow{w}_a(s \leadsto u) - \overrightarrow{w}_a(S_{b(u,d)}) \right)_p$$

$$\cdot \left( w_0(s \leadsto u) + w_0(S_{b(u,d)}) \right) \quad (5)$$

where we have defined $\overrightarrow{w}_a(\overrightarrow{P}) = [w_1(P_1),\ldots,w_m(P_m)]$ and $S_{b(u,d)} = [S_{1(b(u,d))} \ldots S_{m(b(u,d))}]$ with a slight abuse of notation. The value of $E_Q(s \leadsto u)$ is clearly only an approximation of the length of the shortest path which includes $s \leadsto u$, but we can use such value to quickly direct our search toward the shortest path, as explained in the next subsection.

B. Depth-First Search

The look-ahead phase discovers $m+1$ paths $\{S_{i}(s,d)\}_{i=0,\ldots,m}$. If at least one of these paths is feasible, we are given a finite value of $\text{maxlength}$, which represents an upper bound to the shortest path length being the length of the shortest path found so far. But, this may not be the case or the returned value of $\text{maxlength}$ may be a loose upper bound. However, it is desirable that the algorithm finds a feasible path with a length close to that of the shortest path as soon as possible. This is mainly needed for two reasons:

1) the closer $\text{maxlength}$ is to the length of the shortest path the more efficient is the search space reduction. We recall that sub-paths with predicted length larger than $\text{maxlength}$ can be discarded. Thus, the smaller $\text{maxlength}$ the more sub-paths are discarded;

2) if we quickly find a good approximation of the shortest path, it is possible to return such sub-optimal path to reduce computation time.

To achieve the desired objective, look-ahead alone may not be sufficient, as explained above. For this reason, Q-BATE is based on a depth-first search which uses the predicted length information to quickly find a good approximation of the shortest path. Compared to the breath-first search, the depth-first search is usually faster in reaching the destination for the first time. The problem is that a depth-first search may take a long time to return the shortest path if no additional information is employed. We rely on the predicted length information provided by the look-ahead phase to direct the search toward a path with a short length.

C. Pseudo-Code

The pseudo-code of Q-BATE is shown in Fig. 2. The inputs of the Q-BATE procedure are the network graph $G$, the flow of the shortest path found so far. But, this may not be the case or the returned value of $\text{maxlength}$ may be a loose upper bound. However, it is desirable that the algorithm finds a feasible path with a length close to that of the shortest path as soon as possible. This is mainly needed for two reasons:

1) the closer $\text{maxlength}$ is to the length of the shortest path the more efficient is the search space reduction. We recall that sub-paths with predicted length larger than $\text{maxlength}$ can be discarded. Thus, the smaller $\text{maxlength}$ the more sub-paths are discarded;

2) if we quickly find a good approximation of the shortest path, it is possible to return such sub-optimal path to reduce computation time.

To achieve the desired objective, look-ahead alone may not be sufficient, as explained above. For this reason, Q-BATE is based on a depth-first search which uses the predicted length information to quickly find a good approximation of the shortest path. Compared to the breath-first search, the depth-first search is usually faster in reaching the destination for the first time. The problem is that a depth-first search may take a long time to return the shortest path if no additional information is employed. We rely on the predicted length information provided by the look-ahead phase to direct the search toward a path with a short length.
to be routed \( F = (s, d, Q) \) and an integer \( K \) representing the maximum number of times the destination can be reached (i.e. the maximum number of paths to be evaluated). Clearly, only if \( K \) is infinite we are guaranteed that the exact solution is returned. At any time, there exists only one sub-path that \( Q \)-BATE is trying to extend toward the destination. The nodes of such sub-path are stored onto a stack \( S \). Each of such nodes is associated with a priority queue (the \( \text{min\_queue} \) field) containing its neighbors that cannot be excluded based on look-ahead information and that are not present in the stack \( S \). This last property assures that only loop-free sub-paths are considered. To recognize already visited nodes, it suffices to set the \( \text{visited} \) field of the node that is being pushed onto the stack.

The algorithm is initialized by setting the \( \text{visited} \) field of every node to false and emptying the associated priority queue \( \text{min\_queue} \) \(^3\) (lines 3–5). The \( \text{LOOK\_AHEAD} \) procedure is then invoked (line 6), which returns \(-1\) if no path can be found between \( s \) and \( d \), \( \infty \) if all the paths \( \{S_i(s,d)\}_{i=0,\ldots,m} \) are unfeasible and the length of the shortest feasible path among them otherwise. Then, the source node \( s \) is pushed onto the stack \( S \) after setting its \( \text{visited} \) field to true and its \( \text{partial\_weight} \) fields to zero (lines 9–13). The \( \text{partial\_weight} \) \( (i = 0, \ldots, m) \) field of a node \( u \) on the stack represents the \( i \)-th link weight of the current sub-path from \( s \) to \( u \). The procedure \( \text{CREATE\_QUEUE} \) is also invoked to insert all the source’s neighbors that cannot be excluded based on look-ahead information into the priority queue of \( s \).

Then the algorithm enters a while loop which is repeated until either the stack is empty (all the possible paths have been explored) or \( k \) (a counter increased every time the destination has been reached) exceeds \( K \). In every iteration, if the priority queue associated with the node on top of the stack is empty (all of its neighbors have been explored) such node is popped and its \( \text{visited} \) field is set to false because it does not belong to the current sub-path anymore (lines 15–17). Otherwise, a node, say it \( v \), is extracted from the priority queue of the node on top of the stack. The partial weights of \( v \) are obtained starting from the weights of the node on top of the stack. Then, node \( v \) is pushed on top of the stack (lines 18–21). If \( v \) is not the destination node, its \( \text{visited} \) field is set to true and its priority queue is populated with its neighbors. Otherwise, it means a new path has been found. If the length of the new path is smaller than \( \text{maxlength} \) then a shorter path has been found and therefore the new path is stored in \( \text{Path} \) and \( \text{maxlength} \) is updated.

The \( \text{LOOK\_AHEAD} \) procedure (Fig. 3) performs some pre-computation to make available for each node information about the remaining sub-path toward the destination. The \( \text{LOOK\_AHEAD} \) procedure returns the value of the \( \text{maxlength} \) variable, which is initialized to infinity. Links with insufficient available bandwidth are pruned by setting their metric to infinity (lines 2–6). Then, for every link weight the shortest path from each node to the destination is computed. This is accomplished by \( m + 1 \) Dijkstra’s computation of the shortest path tree rooted at the destination on the mirror topology, i.e. the topology obtained by reversing link directions (lines 7–10). As a result of each Dijkstra’s computation, every node is associated with a distance \( \text{dist}(u) \) from the destination \( d \) and a predecessor \( \text{pred}(u) \) along the shortest path to \( d \). For each node \( u \), the distance calculated at the \( i \)-th iteration represents \( w_i(S_i(u),d) \) and is stored in the field \( \text{lookahead}_i \) (lines 11–12). As a consequence of pruning links with insufficient bandwidth it may happen that the network topology becomes a disconnected graph, with \( s \) and \( d \) belonging to different connected components. If this is the case, no feasible path between \( s \) and \( d \) can be found. This event can be identified by looking at the value of \( \text{pred}(s) \). If \( s \) has no predecessor, then it is not reachable from \( d \). In this case (lines 13–14) the procedure \( \text{LOOK\_AHEAD} \) returns \(-1\), which causes the procedure \( \text{Q\_BATE} \) to exit immediately. We note that early identifying the absence of feasible paths due to unavailable bandwidth is another advantage of the look-ahead phase. Only the path from \( s \) to \( d \) is traced by using the predecessor information (lines 15–18). If such path is feasible and has the shortest length among the paths found so far, then the \( \text{maxlength} \) value is updated and the path is stored in the \( \text{Path} \) variable.

The procedure \( \text{CREATE\_QUEUE} \) is invoked when a new node \( u \) is pushed onto the stack \( S \) to select the neighbors to be inserted in its priority queue. For each neighbor \( v \), the procedure \( \text{CREATE\_QUEUE} \) checks whether (i) \( v \) has already been visited (line 2), (ii) link \( u \to v \) has insufficient bandwidth (line 4), (iii) inequality (4) is satisfied (lines 6–7), (iv) one of the inequalities (3) is satisfied (lines 9–13). As soon as one of these conditions is found to be true, the neighbor \( v \) is disregarded. Otherwise (lines 15–16) neighbor \( v \) is inserted.

---

\(^3\)Using the same pseudo-code rules as in [15], a particular field is accessed using the field name followed by the name of its object in square brackets. E.g. \( \text{visited}[u] \) refers to the \( \text{visited} \) field of the node \( u \).
in the priority queue of \( u \) with a key equal to the predicted length (5) of the current sub-path \( s \sim u \) extended toward \( v \). We recall that the procedure \textsc{Extract-Min} used in Q-BATE extracts the node with the minimum key in the priority queue.

\section*{D. Path Length}

The path length function (2) offers three degrees of freedom: \( p \), \( f(\cdot) \) and \( w_0(\cdot) \). We now introduce our choices used in the performance analysis presented in the next section. First, we only consider 2-norms \( (p = 2) \), so we omit the subscript and simply write \(| \cdot |\). As explained in Section V-A, it is possible to further reduce the search space if \( f(\cdot) \) is such that \( f(x) \geq 1 \ \forall x \geq 0 \). We designed function \( f(\cdot) \) according to this requirement and the principle that big distances between \( \tilde{Q}_a \) and \( \tilde{w}_a(P) \) should be penalized:

\[
 f(x) = 2 - \cos \left( \frac{\pi}{2} \frac{x}{|\tilde{Q}_a|} \right) \tag{6}
\]

The function \( f(\cdot) \) so defined is plotted in Fig. 5 as a function of the normalized variable \( x \). We remind we are interested in

\[
 f \left( \left| \tilde{Q}_a - \tilde{w}_a(P) \right| \right) \text{ and } 0 \leq \left| \tilde{Q}_a - \tilde{w}_a(P) \right| \leq |\tilde{Q}_a| \]

for feasible paths, so we are confined to the interval \((0 \ldots 1)\) of values of the normalized variable \( \frac{x}{|\tilde{Q}_a|} \). We note that \( f\left( \left| \tilde{Q}_a \right| \right) = 2 \), thus we can interpret the first component of path length (2) as a multiplicative coefficient ranging from 1 to 2.

As far as the link weight \( w_0(l) \), here we propose a simplified version (as it involves just one parameter) of the function used in SAMCRA-B [3]:

\[
 w_0(l) = \Gamma(\rho_l) = \frac{1}{C(l)} + \delta \frac{\rho_l}{1 - \rho_l} \tag{7}
\]

\( \Gamma(\rho_l) \) is depicted in Fig. 6 as a function of the link utilization \( \rho_l \). The term \( \frac{1}{C(l)} \) diversifies the weight of unloaded links based on their capacity (rather than just assuming the same weight for all the links), while the parameter \( \delta \) influences the slope of the curve. For our simulation, we used \( \delta = 0.5 \).

\section*{VI. PERFORMANCE EVALUATION}

In our experiments, we used two router-level models for generating topologies, Barabasi-Albert and Waxman. All the considered topologies have 100 nodes and a different number of links per new node. For each topology, 10 nodes are randomly chosen to act as edge routers, the entry and exit points for the network traffic, while the other nodes represent core routers, which carry transit traffic only. The capacity of the links is uniformly distributed between 100 and 1024 units. We considered two additive QoS constraints \( (m = 2) \), which have different distributions from one scenario to another. All links are symmetric, with respect to both capacity and QoS link weights.

For all the presented simulations, source and destination nodes are chosen uniformly among the set of edge nodes. Each of the considered scenarios was simulated 10 times with different seeds for the random variables. For each of these 10 iterations, the algorithms under evaluation faced the same set of flow requests. Each iteration involved the generation of 120000 flows. The first 20000 were not considered in our analysis, as they represent a warm-up period needed by the network in order to reach a steady state regime.

For each iteration our simulator computes the call blocking rate (CBR) achieved by each algorithm:

\[
 \text{CBR} = \frac{\text{number of rejected flows}}{\text{total number of flows}}
\]
TABLE I
FLOW AND TOPOLOGY PARAMETERS

<table>
<thead>
<tr>
<th>Test</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow inter-arrival time</td>
<td>Exp(1/0.5)</td>
<td>Exp(1/0.5)</td>
<td>Exp(1/0.25)</td>
</tr>
<tr>
<td>Flow duration</td>
<td>Exp(1/175)</td>
<td>Exp(1/175)</td>
<td>Exp(1/150)</td>
</tr>
<tr>
<td>Requested bandwidth</td>
<td>(U(1, 10)) with (P = 0.75)</td>
<td>(U(1, 10)) with (P = 0.75)</td>
<td>(U(80, 10)) with (P = 0.25)</td>
</tr>
<tr>
<td>First QoS constraint</td>
<td>(U(60, 120))</td>
<td>(U(60, 120))</td>
<td>(U(100, 180))</td>
</tr>
<tr>
<td>Second QoS constraint</td>
<td>(U(70, 140))</td>
<td>(U(70, 140))</td>
<td>(U(130, 190))</td>
</tr>
<tr>
<td>Topology model</td>
<td>Barabasi</td>
<td>Barabasi</td>
<td>Waxman</td>
</tr>
<tr>
<td>Links</td>
<td>294</td>
<td>197</td>
<td>200</td>
</tr>
<tr>
<td>(w_1(l))</td>
<td>(U(3, 15)) with (P = 0.5)</td>
<td>(U(8, 20)) with (P = 0.5)</td>
<td>(U(3, 8)) with (P = 0.75)</td>
</tr>
<tr>
<td>(w_2(l))</td>
<td>(U(25, 40)) with (P = 0.5)</td>
<td>(U(30, 45)) with (P = 0.5)</td>
<td>(U(5, 40)) with (P = 0.25)</td>
</tr>
</tbody>
</table>

Fig. 7. Comparative analysis results
It also computes the throughput after the processing of each new flow request as the sum of the bandwidth requested by the flows crossing the network at that time. In order to get a smooth throughput curve, we first compute the mean over each window of 5000 throughput samples for each iteration and then the average of the corresponding values obtained from the 10 iterations. Finally, we measured the average processor time spent by each algorithm to select a path.

A. Comparative analysis

The purpose of this subsection is to compare different routing algorithms in a dynamic scenario. We evaluated the performance of TE-DB, SAMCRA, SAMCRA-B with $\gamma(\rho_i)$ as link weight $w_0(l)$ and Q-BATE with the definitions of Section V-D. We also wanted to compare these algorithms to MIRA, which is known to achieve a good performance from the viewpoint of resource optimization. MIRA computes a link cost $c_{MIRA}(l)$ for each link $l$ and then uses the Dijkstra’s algorithm to find the shortest path to be returned. Since MIRA does not account for additive QoS constraints, it would not be fair to compare it with the other algorithms. Hence, we compute the link costs $c_{MIRA}(l)$ and then run SAMCRA-B with $w_0(l) = c_{MIRA}(l)$ to find the shortest feasible path. We denote such variant of MIRA by SAMCRA-B(MIRA).

We have carried out a number of simulations using several topologies and loads. In this subsection we illustrate three different scenarios that are representative of the different cases we observed. Table I shows the probability distributions used to characterize the flow requests in such scenarios. The model, the number of links and the link weight distributions of the topologies related to each scenario are also indicated.

By observing the CBR plots (Figg. 7(a), 7(c) and 7(e)), we can conclude that Q-BATE outperforms the other algorithms in the capacity of admitting flows. Indeed, Q-BATE achieves the minimum call blocking rate in all the simulated scenarios. SAMCRA-B($\gamma(\rho_i)$) and SAMCRA-B(MIRA) follow Q-BATE with a gap that is sometimes considerable. TE-DB and SAMCRA, instead, exhibit a poor performance from a dynamic viewpoint.

While the CBR plots show a mean value over all the iterations, the average throughput plots give us information on the average behavior during an iteration. Figures 7(b), 7(d) and 7(f) indicate that the behavior of the algorithms from the viewpoint of throughput is similar to that in terms of CBR. In the sense that the algorithm achieving the minimum CBR also presents the maximum throughput. Throughput plots enable to ascertain that the difference in performance between algorithms is maintained during the whole iteration. Finally, we noticed that Q-BATE requires the minimum path computation time. SAMCRA and SAMCRA-B take a comparable amount of time (about twice and three times that of Q-BATE, respectively), while the time required by TE-DB and SAMCRA-B(MIRA) (based on maxflow computations) is two orders of magnitude larger than that of Q-BATE.

VII. Conclusions

We presented a new algorithm to route flow requests having both a bandwidth requirement and a number of additive QoS constraints. The goal of our algorithm is to minimize blocking while offering QoS guarantees. We proposed a novel path length function to pursue such objective. Q-BATE is based on a look-ahead phase and a depth-first strategy to efficiently find the feasible path with the smallest length. The behaviour of Q-BATE in a dynamic scenario was compared to that of other algorithms. It turned out that Q-BATE outperforms the other algorithms from the viewpoint of admission ratio and throughput.

ACKNOWLEDGMENT

Research outlined in this paper has been partially supported by the European Union under the E-Next Project FP6-506869 and by the Italian Ministry for Education, University and Research (MIUR) in the framework of the QUASAR Project (PRIN Program).

REFERENCES