

Connection Preemption in Multi-Class Networks

Fahad Rafique Dogar, Laeeq Aslam,
and Zartash Afzal Uzmi
Lahore University of Management Sciences
Email: {fahad,laeeq,zartash}@lums.edu.pk

Sarmad Abbasi
National University of Computer
and Emerging Sciences
Email: sarmad.abbasi@nu.edu.pk

Young-Chon Kim
Chonbuk National University
Jeonju, Korea
Email: yckim@chonbuk.ac.kr

Abstract—We address the problem of connection preemption in a multi-class network environment. Our objective is: i) to minimize the number of preempted connections, and ii) to minimize the total preempted bandwidth, in that order. We show that this problem is NP-complete by reducing it to a well-known NP complete problem – the subset sum problem. Therefore, any polynomial time algorithm, such as *Minn_Conn* [1], to solve this problem is suboptimal.

We present an optimal algorithm with exponential complexity that can be used when the network load is light. We also present a fully polynomial time approximation algorithm that performs within a bounded factor from the optimal, and can be used in large networks having thousands of connections. We compare the performance of exact and approximate algorithms in a practical scenario by conducting simulations on a network representing twenty largest metros in the U.S. The simulations show that, on average, the approximate algorithm preempts bandwidth which is only a small fraction more as compared to that preempted by the exact algorithm, but is an order of magnitude more efficient in terms of execution time.

I. INTRODUCTION

Internet is evolving from the traditional best effort service to a multi-class Quality of Service (QoS) aware environment. This is driven by an increasing demand for multimedia applications such as VoIP and video streaming, which has created new challenges for network service providers. One major challenge is to satisfy the QoS requirements of various traffics while judiciously managing the scarce network resources, such as bandwidth. This requires efficient bandwidth reservation and management mechanisms that account for the bandwidth requirements of every request, keeping in view available network resources.

The problem of bandwidth reservation and management in a multi-class network becomes more difficult if requests come in an *online* manner such that no information about future requests is available. Routing of online requests may lead to sub-optimal decisions. For example, a decision to admit the current request may result in insufficient available resources for future high priority requests [2]. On the other hand, it is also undesirable to reject requests, in anticipation of future higher priority requests. *Preemption* allows the flexibility to accept a request and to later retract it, in case a future higher priority request needs resources. This allows efficient use of network resources since bandwidth for higher priority connection requests may be made available through preempting less important connections. Therefore, preemption has become an important component of multi-class QoS provisioning pro-

posals such as DiffServ based Traffic Engineering (DS-TE) [3], [4]. While these proposals allow preemption strategies to be incorporated in earlier traffic engineering approaches, they leave open the choice of *preemption algorithm* that should be used. The preemption algorithm determines the set of connections that should be preempted in order to free up enough resources to accept the new request.

Several preemption algorithms have been proposed that try to optimize different criteria while selecting the set of connections to be preempted [2], [5]–[8]. Common objectives of these preemption algorithms include: minimizing the priority of preempted connections, minimizing the preempted bandwidth, and minimizing the number of preempted connections. Many algorithms try to meet one or a combination of these objectives. Network service providers select the preemption algorithms according to their requirements. This selection depends on several factors such as the cost of rerouting a preempted connection or loss in revenue associated with preempting bandwidth. Therefore, in a practical scenario, priority of preempted connections is not the only important factor: number of preempted connections and the preempted bandwidth also become important.

In this paper, we consider the objective of i) minimizing the number of preempted connections, and ii) minimizing the total preempted bandwidth, in that order. That is, we first minimize the number of connections that need to be preempted and in case there are multiple feasible options, we minimize the total bandwidth of the preempted connections. We show that this problem is NP-complete by reducing it to a well-known NP complete problem – the subset sum problem. Thus, any polynomial time algorithm, such as *Minn_Conn* [1], to solve this problem is suboptimal. A counter example is provided which shows that *Minn_Conn* is, in fact, suboptimal. We provide exact and fully polynomial time approximate algorithms to solve this problem. These algorithms represent the tradeoff between optimal result and execution time. We show through simulations that the approximate algorithm preempts only a small fraction more bandwidth on average compared to the exact algorithm but is an order of magnitude more efficient in terms of execution time.

The rest of the paper is organized as follows: In section II, we define the preemption problem and discuss some earlier work related to this problem. In section III, we formally prove this problem to be NP-complete. Exact and approximate algorithms to solve this problem are presented in section IV

while simulation results comparing their performance are discussed in section V. Finally, we give our conclusions in section VI.

II. PROBLEM BACKGROUND

In this section, we formally define the general preemption problem and discuss earlier work related to the specific problem that we are addressing in this paper.

A. Problem Definition

We consider a multi-class network which has provision for bandwidth guaranteed paths, similar to the one specified in the DS-TE proposal [3]. Requests arrive in an online manner and no information about future requests is assumed. The requests arrive at a source node, which calculates the path to the destination and subsequently reserves resources along this computed path. The calculation of the path is done by using a constraint-based approach, such that the bandwidth requirements are satisfied. While calculating the path all bandwidth reserved for lower classes is included as available bandwidth,¹ with the assumption that this bandwidth, currently belonging to lower class traffic, will be available in case a preemption is needed. If a feasible path is found then the request is accepted, otherwise the request is rejected.

In case a request is accepted, the computed path is signalled through resource reservation protocols such as RSVP-TE. Each node along this path makes bandwidth reservations for its outgoing link, completing the path reservation in a distributed fashion. If the residual bandwidth on the link is greater than the bandwidth request then preemption is not required. Otherwise, a preemption algorithm is used to free up enough resources so that the residual bandwidth on the link becomes greater than the bandwidth request. The connections marked for preemption are retracted and bandwidth reservations are made for the new request. The preempted connections may or may not be rerouted depending on the choice and policies of the network service provider.

B. Related Work

The above mentioned distributed approach, where each node has to make preemption decision on its outgoing link, is more suitable for integration with distributed protocols such as OSPF and RSVP. However, we can also have a centralized scenario where a central entity having complete information about the network state makes the preemption decision. The preemption decision has to account for changes in the whole network as a result of preempting any connection. Therefore, the preemption decision must be based on a global view of the whole network. This makes optimization of objectives, such as minimizing the preempted bandwidth or minimizing the number of preempted connections, an NP-complete problem in a centralized scenario [2]. However, considering the distributed nature of Internet protocols, a distributed preemption approach is preferred, and has therefore been the subject of several

recent studies [2], [5]–[7]. These distributed preemption algorithms try to achieve various objectives such as: minimizing the preempted bandwidth, minimizing the number of preempted connections, minimizing the priority of preempted connections or a combination of the above. Note that each node tries to optimize the preemption decision for its outgoing link only regardless of how this decision would affect other preemptions in the networks.

A distributed preemption algorithm *Min_Conn* was proposed by Peyravian et al. in [1] to optimize the following criteria: i) minimizing the number of preempted connections, ii) minimizing the preempted bandwidth, and iii) minimizing the priority of preempted connections, in that order. The *Min_Conn* algorithm, reproduced in Figure 1, runs in polynomial time but it does not provide optimal results as claimed by the authors.

Fig. 1. *Min_Conn* Algorithm

```

1) while  $B_p > a_j$  do
2)    $W := B_p - a_j$ ;
3)    $i = 0$ ;
4)   for  $l = 1$  to  $k$  do
5)     if  $i = 0$  and  $B_l \geq W$  then  $i := l$ ;
6)     if  $i > 0$  and  $B_l \geq W$  and
7)       ( $B_l < B_i$  or ( $B_l = B_i$  and  $P_l < P_i$ )) then  $i := l$ ;
8)   endfor;
9)   if  $i = 0$  then
10)     $i = 1$ ;
11)    for  $l = 1$  to  $k$  do
12)      if  $B_l > B_i$  or ( $B_l = B_i$  and  $P_l < P_i$ ) then
13)         $i := l$ ;
14)    endfor;
15)     $C := C - \{C_i\}$ ;
16)     $P := P \cup \{C_i\}$ ;
17)     $a_j := a_j + B_i$ .
18) endwhile

```

In the *Min_Conn* algorithm, B_p is the bandwidth demand of the new high-priority request on a link which has free residual bandwidth a_j . Preemption will be required on that link only when the bandwidth demand exceeds the residual bandwidth; the excess required bandwidth is W which needs to be freed up by preempting some existing connections. The link is assumed to have a number of existing low-priority preemptable connections represented in a set $C = \{C_1, C_2, C_3, \dots, C_k\}$, where each connection C_i has an associated pair (B_i, P_i) with B_i and P_i representing the bandwidth and priority of the connection, respectively. Note that the *Min_Conn* algorithm is invoked for a link on the computed path where the path computation algorithm ensures that B_p will not exceed the sum of residual and preemptable bandwidth on that link.

The *Min_Conn* algorithm returns P , a set of connections that should be preempted. If all preemptable connections have the same priority, then the algorithm first minimizes the number of preempted connections and then minimizes the preempted bandwidth. In the loop starting at step 4, the algorithm checks for a single request that could satisfy the preemption requirement and selects the one which minimizes the preempted bandwidth. If multiple connections need to be preempted, indicated by a true condition at step 10, then the algorithm adopts a greedy approach and includes the

¹Such information is readily available through extensions to link state routing protocols; see [9] for details.

connection with the largest bandwidth in the preemption step. The procedure is repeated until enough resources are made available to accommodate the new request.

To show that *Min_Conn* is suboptimal, consider a link with $C = \{70, 50, 50, 20\}$, $a_j = 0$, and a new request has a required bandwidth $B_p = 100$. The output of the *Min_Conn* algorithm will be $P = \{70, 50\}$ whereas the optimal result is $P = \{50, 50\}$. Clearly, the greedy approach of selecting the largest bandwidth connection is sub-optimal.

III. PROOF OF NP-COMPLETENESS

We now show that the problem of minimizing the number of preempted connections, and minimizing the preempted bandwidth, in that order, is NP-complete².

A. Formal definitions

Let V is the set of bandwidths of preemptable connections and t is the bandwidth that needs to be preempted. Recall that preemption is only required when the bandwidth of a new high-priority connection exceeds the remaining bandwidth available on the link. We first consider the following problem:

Problem 3.1: Given a set $V = \{a_1, \dots, a_n\}$ of n positive integers and a number t , find a minimum cardinality subset, S , of V such that

$$\sum_{a_i \in S} a_i \geq t$$

This problem has an easy polynomial-time solution. We may first sort the elements of set V in $O(n \log n)$ time such that $a_1 > a_2 > \dots > a_n$, after which we find the minimum K such that $\sum_{i=1}^K a_i \geq t$. The output set is $S = \{a_1, \dots, a_K\}$ if the constraint can be satisfied, otherwise a null set is returned. Next, we consider the following problem:

Problem 3.2: Given a non-null set $S = \{a_1, \dots, a_K\}$ of K positive integers and a number t , is the sum of elements of S equal to t ?

The solution to this problem is trivial. Finally, we formally define the preemption problem we are addressing in this paper (i.e., first find the minimum number of connections to be preempted and then find the actual set of connections to be preempted while minimizing the bandwidth overshoot):

Problem 3.3: Given a set $V = \{a_1, \dots, a_n\}$ of n positive integers and a number t , find a set S such that the following conditions are satisfied:

- i) $\sum_{a_i \in S} a_i \geq t$
- ii) $|S| = K$
- iii) $\sum_{a_i \in T} a_i < t, \quad \forall T : |T| < K$
- iv) $\sum_{a_i \in S} a_i$ is minimized

²This is the special form of *Min_Conn* [1] where the priority of all preemptable connections is assumed to be the same. Proving that the first two criteria make this an NP-complete problem is sufficient since any further criterion would only add to the complexity of the problem.

Solution to this problem, which internally generates K , will be a null set if the threshold constraint (i) cannot be satisfied. When (i) can be satisfied, computing the value of K is not difficult. Indeed, it is the cardinality of the set that solves Problem 3.1.

We show that the solutions to the above three problems can be used to solve the subset sum problem. Since the subset sum problem is known to be NP-complete [10], and Problems 3.1 and 3.2 are solvable in polynomial time, this will lead to the conclusion that Problem 3.3 is NP-complete. The subset sum problem is defined as [10]:

Problem 3.4 (Subset Sum): Given a set $V = \{a_1, \dots, a_n\}$ of n positive integers and a number t , is there a subset S , of V , such that

$$\sum_{a \in S} a = t$$

B. NP-completeness of Problem 3.3

We first provide a polynomial-time construction to show that solutions to Problems 3.1, 3.2 and 3.3 may be used to find a solution to subset sum problem in a polynomial time. Suppose we are trying to solve subset sum problem with inputs $V = \{a_1, \dots, a_n\}$ and t . Without loss of generality, we may assume that each integer in V can be expressed in l bits where l can be arbitrarily large.

We construct $V' = \{b_1, \dots, b_n, c_1, \dots, c_n\}$ from V as a set of $2n$ positive integers where each integer is a $\log n + l + 3n$ bit number. All the bits of integer b_i have a value 0 except the bits at the $(\log n)$ -th and $(\log n + 2i)$ -th position from the left, which have a value 1. We also choose $c_i = b_i + a_i$. Pictorially b_i and c_i are represented as:

$$\begin{array}{c}
 \text{log } n \text{ bits} \quad \overbrace{\quad \quad \quad}^{n \text{ pairs}} \quad \text{log } n \text{ bits} \\
 b_i = \underbrace{00 \dots 01}_{\text{log } n \text{ bits}} \quad \underbrace{00 \dots 00}_{i\text{-th}} \quad \underbrace{01}_{i\text{-th}} \quad \underbrace{00}_{n \text{ zeros}} \quad \underbrace{00 \dots 00}_{l \text{ zeros}} \\
 \\
 \text{log } n \text{ bits} \quad \overbrace{\quad \quad \quad}^{n \text{ pairs}} \quad \text{log } n \text{ bits} \quad a_i \\
 c_i = \underbrace{00 \dots 01}_{\text{log } n \text{ bits}} \quad \underbrace{00 \dots 00}_{i\text{-th}} \quad \underbrace{01}_{i\text{-th}} \quad \underbrace{00}_{n \text{ zeros}} \quad \underbrace{01 \dots 10}_{a_i}
 \end{array}$$

We also construct t' as given in the following picture:

$$t' = \overbrace{011 \dots 010}^{n \text{ (in log } n \text{ bits)}} \quad \overbrace{01 \quad 01 \quad \dots \quad 01}^{n \text{ pairs}} \quad \overbrace{000110 \dots 010}^{t \text{ (in } l + n \text{ bits)}}$$

It is now straightforward to prove that the subset sum problem is solvable in polynomial time if Problem 3.3 is solvable in polynomial time. To show this, we observe that construction of V' and t' from V and t , respectively, is a polynomial-time operation. We can then use the following steps in trying to solve the subset sum problem:

- 1) Invoke Problem 3.1 and if the returned set is null, the subset sum problem is solved (i.e., there does not exist a subset that can satisfy the constraint). No further processing is necessary.

- 2) If the returned set in above step is not null, construct V' and t' from V and t , respectively, and invoke Problem 3.3 with inputs V' and t' . Save the returned set S' .
- 3) Generate a set S'' from S' by keeping only l least significant bits of each element.
- 4) Invoke Problem 3.2 with inputs S'' and t .

We claim that the answer in the last step would be the same as the answer provided by the subset sum problem. To show this, we note the following:

- a) When Problem 3.3 is invoked in step 2 above, the most significant $\log n$ bits of the elements of V' and of t' ensure that the internally generated value of K is at least n . Since the process did not terminate at step 1, the internally generated value of K is at most n . Thus, the invocation of Problem 3.3 in step 2 would result in a set S' whose cardinality is n .
- b) The construction also ensures that one and only one from the pair (b_i, c_i) is returned in S' . If both b_i and c_i are included in S' , then there is at least one other pair (b_j, c_j) such that neither of b_j and c_j are selected. If $j < i$, then the threshold condition will not be satisfied, and if $j > i$, then overshoot is not minimized. Thus exactly one from the pair (b_i, c_i) is selected in S' .
- c) An element c_j is included in S' if and only if a_j would have been included in the result obtained from Problem 3.3 with inputs V and t .
- d) The set S'' contains trimmed versions of elements included in S' . For any value of i , the number obtained by trimming b_i to l least significant bits is 0. On the other hand, the number obtained by trimming c_i to l least significant bits is a_i . Thus, summation of elements of S'' is exactly the same as obtained by summing the elements of a set obtained by invoking Problem 3.3 with inputs V and t .
- e) The final step where Problem 3.2 is invoked simply results in the solution of subset sum problem.

In summary, we have shown that a solution to subset sum problem can be obtained in polynomial time by using solutions to Problems 3.1, 3.2 and 3.3. Since the subset sum problem is known to be NP-complete and Problems 3.1 and 3.2 are solvable in polynomial time, we conclude that Problem 3.3 is NP-complete.

IV. ALGORITHMS

In this section, we present exact and approximate algorithms to solve Problem 3.3. We use the exposition used in [10] for solving the subset sum problem. As mentioned in the last section, finding the minimum number of connections for preemption, K , is possible in $O(n \log n)$ steps. Therefore, for both the algorithms we provide K as input, in addition to the preemptable set of connections, $V = \{a_1, \dots, a_n\}$, and the bandwidth required for preemption t .

A. Exact Algorithm

The proposed exact algorithm, shown in Figure 2, consists of n iterations. In each iteration i , a_i is added to each element of another list L . Therefore the list L , at any iteration i , maintains all the possible sum values that can be obtained by adding any combination of elements from the set $\{a_1, \dots, a_i\}$. We initialize list L with 0 so that each element a_i , when added with zero, also becomes part of the list. A counter is maintained for each sum value in L which keeps track of the number of elements that add up to make that particular sum value.

Fig. 2. Exact algorithm (V, t, K)

```

1)  $L_0 := 0$ 
2)  $SolutionSum := \infty$ 
3) for every element  $a_i$  in  $V$ 
4)   Add  $a_i$  to every element of  $L$  and
5)   insert the sum values in sorted order in  $L$ 
6)   if duplicate then keep element with smaller counter
7)   for every element  $L_j$  in  $L$ 
8)     if  $(L_j > t$  and  $L_j < SolutionSum)$ 
9)       then  $SolutionSum := L_j$ 
10)    if  $Counter(L_j) > K$  or  $L_j \geq SolutionSum$ 
11)      then discard element

```

Steps 4-6 of the exact algorithm perform the process of adding new sum values to list L in sorted manner. Note that if there are multiple elements with the same sum value then we only keep the element with the lowest counter value. We only need to keep this value since we are assured that the threshold t cannot be exceeded with $K - 1$ elements. Once the new sum values are added to L , we completely traverse L and discard all those elements whose counter values are greater than K . This ensures that all those elements whose value exceeds the threshold have a counter value of exactly K . Such values are compared with $SolutionSum$ which keeps track of the least bandwidth that exceeds the threshold t . Either we update $SolutionSum$ or we discard the element. At the end of the algorithm, $SolutionSum$ gives a solution to our problem. Since the length of L , in any iteration i , can be as much as 2^i , the exact algorithm runs in exponential time.

B. Approximate Algorithm

Note that the exact algorithm given above runs in exponential time because the worst-case length of L is exponential in n . Our approximate algorithm is the same as the exact algorithm except that it ensures that the size of list L remains polynomial in n . An algorithm called *Trim* is used to constrain the length of list L after step 6 in exact algorithm. *Trim* is based on the idea that if two values are quite close, then for the purpose of approximation there is no need to store both the values: we can always store only the larger value which can act as a representative of the smaller value. This ensures that our solution, although not optimal, is still feasible since the optimal value can only be replaced by a larger value.

The *Trim* algorithm, listed in Figure 3, takes as input the monotonically increasing list L and a trim factor δ . We define δ in such a way that if L' is the trimmed list, then for every

element y that was removed from L , there is an element z , with the same counter value, still in L' that approximates y , such that $y \leq z \leq y(1 + \delta)$. The *Trim* algorithm keeps record of the last value stored for each counter in an array, *Last*, which is indexed by the counter values.

Fig. 3. *Trim*(L, δ) returns L'

1)	$m := L $
2)	$L' := y_m$
3)	$Last := \infty$
4)	$Last[counter(y_m)] := y_m$
5)	for $m - 1$ to 1
6)	if $y_m(1 + \delta) < Last[counter(y_m)]$
7)	append y_m to the start of L'
8)	$Last[counter(y_m)] := y_m$

Trim ensures that, for any non-zero counter value, there are at most $\lfloor 1 + \log_{1+\delta} 2t \rfloor$ elements in L' .³ Since L is initialized with 0 and there are at most K non-zero counter values represented in L , the total number of elements in L' are at most $1 + K \lfloor (1 + \log_{1+\delta} 2t) \rfloor$ with complexity $O(\frac{K \log t}{\delta})$. This makes the list size polynomial in the size of the input. We further note that the approximate algorithm runs in $O(\frac{nK \log t}{\delta})$ time.

Each time *Trim* is called, the maximum departure from the optimal solution is limited to a factor $1 + \delta$. Since the counter value never exceeds K , the maximum accumulated deviation from the optimal solution is limited to a factor $(1 + \delta)^K$. That is, if S is the optimal solution then, the approximate algorithm is guaranteed to output a solution S^* such that $\frac{S^*}{S} \leq (1 + \delta)^K$. A trivial solution can be constructed for $\delta \geq 1$, therefore, we only consider $\delta < 1$ for which $\frac{S^*}{S} \leq (1 + \delta)^K < 1 + 2K\delta$. In order to guarantee a solution that is within a factor of ϵ , we can take $\delta = \frac{\epsilon}{2K}$. In this case, the running time becomes $O(\frac{nK^2 \log t}{\epsilon})$, thereby making our approximation algorithm a fully polynomial time approximation scheme.

C. Example

We now present an example with $V = \{40, 45, 50, 55, 80\}$ and $t = 140$ to illustrate the working of above algorithms. We show the list L and *SolutionSum* at the end of each iteration as a collection T . The subscript with the sum values in L represents their counter value. The sums L_j^* represent those values that are removed from list L during the iteration but are shown only for illustration.

First, for the exact algorithm, the collection T at each iteration is shown in Figure 4. The exact algorithm gives a solution of 140 which is exactly equal to the required bandwidth t .

The working of approximate algorithm that uses *Trim* with $\delta = 0.2$ is shown in Figure 5. The sums L_j^t represent those values that are trimmed during this iteration and subsequently eliminated from the list but are shown only for illustration.

³This assumes that no element in the original list V is greater than or equal to t in which case the solution is trivial. Thus, for any given non-zero counter value, the element values in L' may not exceed $2t$ and, additionally, these values must be apart by a minimum factor of $1 + \delta$.

Fig. 4. Example: Exact Algorithm

i	T
0	$\{\{0_0\}, 0\}$
1	$\{\{0_0, 40_1\}, 0\}$
2	$\{\{0_0, 40_1, 45_1, 85_2\}, 0\}$
3	$\{\{0_0, 40_1, 45_1, 50_1, 85_2, 90_2, 95_2, 135_3\}, 0\}$
4	$\{\{0_0, 40_1, 45_1, 50_1, 55_1, 85_2, 90_2, 95_2, 100_2, 105_2, 135_3, 140_3^*, 145_3^*, 150_3^*, 190_4^*\}, 140\}$
5	$\{\{0_0, 40_1, 45_1, 50_1, 55_1, 80_1, 85_2, 90_2, 95_2, 100_2, 105_2, 120_2, 125_2, 130_2, 135_2, 165_3^*, 170_3^*, 175_3^*, 180_3^*, 185_3^*, 215_4^*\}, 140\}$

Fig. 5. Example: Approximate Algorithm

i	T
0	$\{\{0_0\}, 0\}$
1	$\{\{0_0, 40_1\}, 0\}$
2	$\{\{0_0, 40_1^t, 45_1, 85_2\}, 0\}$
3	$\{\{0_0, 45_1^t, 50_1, 85_2^t, 95_2, 135_3\}, 0\}$
4	$\{\{0_0, 50_1^t, 55_1, 95_2^t, 105_2, 135_3^t, 150_3^*, 190_4^*\}, 150\}$
5	$\{\{0_0, 55_1, 80_1, 105_2, 135_2, 185_3^*\}, 150\}$

The solution obtained through the approximate algorithm is 150. Note that *Minn_Conn* would adopt a greedy approach, thereby selecting 80 and 55 as the first two connections. For the last connection, *Minn_Conn* would try to minimize the overshoot and would therefore select 40. So the result of *Minn_Conn* algorithm would be 175, compared to 140 and 150 for exact and approximate algorithms, respectively.

V. SIMULATION EXPERIMENTS

In this section, we describe the simulation experiments used to compare the exact and approximate algorithms. Our experiments depict the tradeoff between an exact solution and the time taken for computation. To this end, we present results that show the extra bandwidth that is preempted because of approximation. Similarly, we also show the performance benefit gained, in terms of execution time,⁴ when using the approximate algorithm. Our results show that the approximate algorithm has significantly lower execution time while preempting little extra bandwidth when compared to the exact algorithm.

The simulation experiments are conducted on a homogeneous network topology which is adapted from the network used in [11]. It represents the Delaunay triangulation for the twenty largest metros in continental United States [11]. All links in the network are uni-directional having a capacity of 48 units. The bandwidth constraint model used for link advertisement and reservation is the Russian Doll bandwidth constraint model [12]. There are two QoS classes: high priority and low priority with bandwidth constraints 50% and 100%, respectively.

The traffic matrix consists of 10,000 connections, where 20% of the requests are of high priority class while the rest are of low priority. Requests arrive with a constant inter-arrival time of one unit⁵ and are characterized by a source, a destination, traffic class, the associated bandwidth, and the holding time of the request. The source and destination nodes are chosen randomly from amongst all source-destination pairs.

⁴The execution time was measured on a Pentium IV, 2.8 GHz machine with 1 GB RAM.

⁵Total simulation time is 10,000 units.

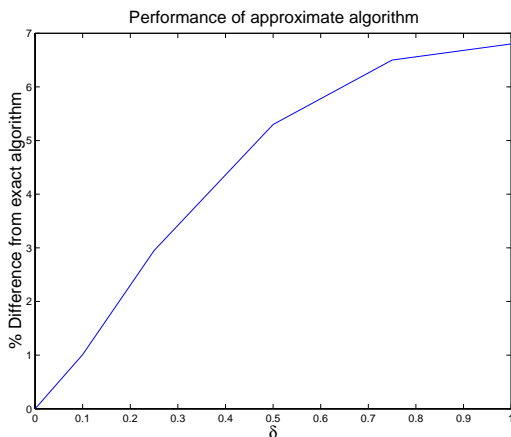


Fig. 6. Performance of approximate algorithm for different values of δ

The bandwidth demand for a request is uniformly distributed between 21 and 24 units for the high priority class and between 0.1 and 15 units for lower priority class. Finally, the call holding time for each request is uniformly distributed between 300 and 800 units.

Fig. 7. Comparison of Execution Time

Algorithm	Execution Time (sec)
<i>exact</i>	2137
<i>approx</i> ($\delta = 0.001$)	134
<i>approx</i> ($\delta = 0.01$)	64
<i>approx</i> ($\delta = 0.1$)	59
<i>approx</i> ($\delta = 1.0$)	58

With the above topology and traffic matrix, we generate results to compare the performance of exact and approximate algorithms. To run the approximate algorithm we have to choose an approximation parameter ϵ and set $\delta = \frac{\epsilon}{2k}$. Instead, we run the approximate algorithm for different values of δ from 0 to 1. Our simulation results show that output of these algorithms give significantly improved results than the ones guaranteed by the worst case analysis of section IV-B. Setting the value of δ to a fixed constant independent of n has the positive effect of reducing the running time to $O(nk \log t)$ and thus making the algorithm more practical.

Figure 6 shows the deviation exhibited by the approximate algorithm from the exact algorithm, in terms of preempted bandwidth, for different values of δ . For generating this result, while we preempt connections according to the exact algorithm, we also compute the bandwidth that would have been preempted by the approximate algorithm for each preemption case. The results show that as we increase δ , the error caused by approximation increases. The extra bandwidth preempted for $\delta = 0.1$ is only about 1% and goes up to approximately 6.5% for $\delta = 1.0$.

In another simulation, we separately run the exact and ap-

proximate algorithms and compare their total execution time. Figure 7 shows this comparison, which clearly indicates the exponential nature of the exact algorithm compared to the approximate algorithm. The exact algorithm takes approximately 2137 seconds which is significantly higher than the execution time of approximate algorithm. Results of approximate algorithm for different values of δ show that the average execution time becomes almost constant for $\delta \geq 0.01$. This indicates that for a given traffic matrix, the improvement in execution time gained through increasing δ , becomes negligible after a certain value of δ is reached.

VI. CONCLUSIONS

In this paper, we discussed the problem of minimizing the number of preempted connections and minimizing the preempted bandwidth, in that order. We proved that this problem is NP-complete by reducing it to the subset sum problem. We also showed that *Minn_Conn*, which claims to solve this problem in polynomial time, gives sub-optimal results. We proposed exact and fully polynomial approximation algorithms and performed simulations to compare their performance under practical conditions. Our simulations show that, on average, the approximate algorithm preempts bandwidth which is only a small fraction more as compared to that preempted by the exact algorithm, but is an order of magnitude more efficient in terms of execution time. Therefore, it can be used for large networks having thousands of connections.

REFERENCES

- [1] M. Peyravian and A. D. Kshemkalyani, "Connection Preemption: Issues, Algorithms, and a Simulation Study," in *Proceedings of Infocom*, 1997, pp. 143–151.
- [2] J. A. Garay and I. S. Gopal, "Call Preemption in Communication Networks," in *Proceedings of Infocom*, 1992, pp. 1043–1050.
- [3] F. L. Faucheur, L. Wu, B. Davie, S. Davari, P. Vaananen, R. Krishnan, P. Cheval, and J. Heinanen, "RFC 3270: Multi-protocol Label Switching (MPLS) Support for Differentiated Services," May 2002.
- [4] —, "RFC 3564: Requirements for support of differentiated services-aware MPLS traffic engineering," July 2003.
- [5] J. D. Oliveira, C. Scoglio, I. Akyildiz, and G. Uhl, "New Preemption Policies for DiffServ-Aware Traffic Engineering to Minimize Rerouting in MPLS networks," in *Transactions on Networking*, vol. 12, no. 4, August 2004, pp. 733–745.
- [6] S. Jeon, R. T. Abler, and A. E. Goulart, "The Optimal Connection Preemption Algorithm in a Multi-Class Network," in *Proceedings of ICC*, 2002, pp. 2294–2298.
- [7] L. Lei and S. Sampalli, "Backward Connection Preemption in Multi-class QoS-aware Networks," in *Proceedings of ICC*, 2004, pp. 153–157.
- [8] S. Poretzky and T. Ganon, "An Algorithm for Connection Precedence and Preemption in Asynchronous Transfer Mode (ATM) Networks," in *Proceedings of ICC*, 1998, pp. 299–303.
- [9] F. L. Faucheur(Editor), "Internet Draft: Protocol Extensions for Support of Differentiated-Service-Aware MPLS Traffic Engineering," March 2004.
- [10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," 2001.
- [11] S. Norden, M. M. Buddhikot, M. Waldvogel, and S. Suri, "Routing Bandwidth Guaranteed Paths with Restoration in Label Switched Networks," in *Proceedings of ICNP*, November 2001, pp. 71–79.
- [12] F. L. Faucheur, "RFC 4127: Russian Doll Bandwidth Constraint Model for DiffServ-aware MPLS Traffic Engineering," June 2005.