Routing Multiple Unsplittable Flows Between Two Cloud Sites with QoS Guarantees

Erdal Akin, Turgay Korkmaz
University of Texas at San Antonio, Texas, USA
Email: erdal.akin@utsa.edu, korkmaz@cs.utsa.com

***Abstract*—Large scale cloud applications may require users to get multiple resources (e.g., VMs, storages) from different sites and simultaneously connect each pair of resources by a path that can satisfy certain Quality-of-Service (QoS) requirements. Finding such paths with the bandwidth constraint is known as the unsplittiable flow problem, which is shown to be NP-hard. Accordingly, various approximations as well as heuristic algorithms are proposed to maximize the amount of sent flows. However, the existing solutions suffer from either low performance in maximizing the amount of sent flows or excessive computation time. In this paper, we propose a new efficient heuristic for a special case, where multiple flows are sent between two cloud sites. Our heuristic algorithm first determines level cuts between the source and destination sites. It then rearranges the flow set, if total bandwidth requirement exceeds total bandwidth of minimum level cut and assigns the flows to the links in the possible bottleneck cuts using randomized best fit approach. It finally tries to forward flows between the cuts using bandwidth constrained shortest path algorithm per flow. We demonstrate the efficiency of our heuristic using simulation.**

***Index Terms*—Cloud Computing, SDN, QoS, Multi-Flow, Bandwidth constraint path selection**

I. Introduction

Cloud Computing is a new paradigm that allows users to access a shared pool of resources such as servers, storage, and applications from anywhere [1], [2]. Because of its flexible and economically advantageous model, many institutions and companies have been increasingly using the cloud for offering various traditional services (e.g., storage, e-mail). However, to support large scale and more complicated services involving multiple resources from different cloud sites (e.g., content distribution and replication, fault tolerance), we need to figure out how to simultaneously connect multiple resources in different cloud sites through an underlying cloud network that can provide Quality-of-Service (QoS) guarantees [3].

To satisfy the QoS requirements of the multiple flows, we would like to consider bandwidth as the main QoS parameter. Accordingly, the problem we study becomes very much related to bandwidth-constrained path problems [4], [5] and maximum multi-commodity flow problems [6]. Bandwidth-constrained path selection has been extensively studied in the literature and shown to be an easy problem in the case of a single flow [7]. However, in the case of multiple flows, the problem turns out to be NP-hard as it is related to the integer multi-commodity flow problem in the literature [6], [8]–[10]. More specifically, it is known as the Unsplittable Flow Problem (UFP) and can formally be defined as follows.

Definition 1: **Unsplittable Flow Problem (UFP)** : Consider a network that is represented by a directed simple graph $G = (N, A)$, where N is the set of nodes, $N = \{v_1, v_2, v_3, ..., v_n\}$ and A is the set of links, $A = \{e_1, e_2, e_3, ..., e_m\}$. Each link $(i, j) \in A$ is associated with an available bandwidth parameter $bw(i, j) \geq 0$. A flow f_k is a 3-tuple (s_k, d_k, r_k) where s_k is source node, d_k is destination node and r_k is the bandwidth requirement of flow f_k. Each flow f_k needs to be routed through a single path p_k for which $bw(p_k) \stackrel{\text{def}}{=} \min\{bw(i, j)|(i, j) \in p_k\} \geq r_k$. The problem is to find paths that can maximize the total routed demand from given K flows $F = \{f_1, f_2, f_3, ..., f_K\}$:

$$max \sum_{k=1}^{K} r_k y_k$$

subject to

$$\sum_{k=1}^{K} r_k x_{ij}^k \leq bw(i, j) \; for \; each (i, j) \in A.$$

$$\sum_{j:(i,j)\in A} r_k x_{ij}^k - \sum_{j:(j,i)\in A} r_k x_{ji}^k = \left\{ \begin{array}{ll} +r_k & if \; i = s_k \\ -r_k & if \; i = d_k \\ 0 & otherwise \end{array} \right\} \forall i, k$$

$$x_{ij}^k = \left\{ \begin{array}{ll} 0 & : x_{ij}^k \notin p_k \\ 1 & : x_{ij}^k \in p_k \end{array} \right.$$

$$y_k = \left\{ \begin{array}{ll} 1 & : \exists \; p_k \;\; s.t. \;\; r_k \leq bw(p_k) \\ 0 & \qquad\qquad Otherwise \end{array} \right.$$

Even in a special case, where all the flows with different bandwidth constraints are between the same source and destination nodes, the problem is still NP-Hard [11]. In this paper, we specifically focus on this special case. As a motivating application, we can consider replicating multiple real time multimedia streams from one cloud site to another, or backing up data for multiple tenants between two cloud sites.

Since there is no polynomial algorithm for the problem unless P=NP [10], researchers have proposed various heuristics and approximation algorithms as we review in the next section. The main problem with the existing algorithms is that they suffer from excessive computation time when trying to provide good performance in maximizing the amount of sent demand. In this paper, our goal is to provide a new computationally

efficient heuristic for the special case requiring to find multiple bandwidth-constrained paths between *two* cloud sites.[1]

In essence, the existing simple heuristics use a traditional bandwidth-constrained path selection algorithm (e.g., widest-shortest path [7]) in an *iterative* manner. Basically, after sorting the flows in some order (e.g., ascending, descending, random w.r.t. bandwidth requirements), the iterative heuristics try to find a path for each flow. However, as we demonstrate later, this iterative approach often fails because the paths that are found first often block the bottleneck links and prevent finding other paths that can send more flows. To overcome this blocking problem, the authors in [16] propose a heuristic called Greedy Algorithm with Preemption (GAP). It mainly preempts previously connected paths and identifies new paths that can maximize the total routed demand. While this approach performs well in maximizing the total routed demand, it suffers from excessive computation time as it repeatedly preempts previously connected paths and tries new ones.

In contrast, our heuristic adopts a different approach where it first determines level cuts between the source and destination sites. It then rearranges flow set by eliminating some flows using modified knapsack algorithm with randomization, if total requested demand is greater than total bandwidth of minimum level cut. Then, it determines possible bottleneck cuts and assigns flows to links in those cuts using randomized best fit approach. Finally, for each flow, it tries to find a bandwidth constrained path through the links that are assigned to carry the flow. Using simulation we demonstrate that our heuristic achieves the similar or better performance in maximizing the total routed demand while significantly reducing the computation time when compared to the recently proposed effective heuristic in [16].

The rest of this paper is organized as follows. In Section II we present the related work. We describe our proposed heuristic algorithm in Section III. In Section IV we present our simulation setup and the results. Finally, we conclude this paper and discuss some possible future work in Section V.

II. Related Works

The problem we study is very much related to bandwidth-constrained path problems [4], [5] and maximum multi-commodity flow problems [6]. Existing bandwidth-constrained path selection algorithms are mainly consider a *single flow* and thus can easily be solved. One of the best known traditional routing algorithm is Widest-Shortest Path algorithm [7]. The algorithm tries to find the path with largest bandwidth throughput while minimizing hop-count. Another version known as Shortest-Widest Path algorithm first eliminates all the links that does not support the given bandwidth requirement and then executes the Dijkstra's shortest path algorithm which selects the links with largest bandwidth capacity in case of cost equality [17].

Existing solutions to maximum multi-commodity flow problems [6] often assume that the flows can be divided and sent over different paths. However, in our case, each flow should follow a single path to avoid the problems related to re-ordering of the packets. Accordingly, the problem becomes the Unsplittable Flow Problem (UFP). It is firstly addressed in [11]. Under the assumption that maximum demand of a commodity, r_{max}, has to be less than the minimum link capacity, which is known as *no-bottleneck* assumption, researchers have developed various approximation algorithms, which often suffer from computational complexity while being hard to implement.

In contrast, the researchers have also explored the possibility of developing easy-to-implement but effective heuristics with no assumptions, as done in this paper. In this direction, the first heuristic called Greedy Algorithm (GA) is introduced in [11]. GA is an iterative algorithm that attempts to find a shortest path for each request. It tries each request once and accepts it if there is a feasible path; otherwise, rejects it. To improve its performance, the authors in [18] has modified GA and called it Bounded Greedy Algorithm (BGA). In essence, BGA limits hop counts with a bound, L, when trying to find a path for each flow as in GA. The authors have also proposed further modifications and called it Careful Bounded Greedy Algorithm (cBGA). In addition to the hop count bound (L), cBGA considers another bound to limit the overloading link capacities of a path while searching a feasible path for each request. In [16], the authors have mainly considered the ideas behind the above iterative solutions and developed a new heuristic, called Greedy Algorithm with Preemption (GAP). It basically tries to find better solutions by preempting an existing path and re-trying the flows failed previously. Overall, this new approach gives the best performance in terms of maximizing the amount of sent data. Thus, we will compare our solution against this heuristic in [16] and show that our heuristic significantly reduce computation time while also achieving slightly better performance in maximizing the amount of sent data.

III. Proposed Heuristic Algorithm

The simplest heuristic is to use the existing path selection algorithms (e.g., widest-shortest path or shortest-widest path) in an *iterative* manner [11], [16], [18]. For this, we can consider the given flows in some order (e.g., ascending, descending, or random order w.r.t. bandwidth requirements) and try to find a path for each flow f_k using, for example, the

[1]While focusing on how to solve this challenging problem, we assume that the underlying network state information (e.g., available bandwidth of each link) is accurately collected (e.g., using OSPF [12]), and the network is able to establish per-flow connections by allocating bandwidth resources. As a matter of fact, Software Defined Networking (SDN) has recently become a popular paradigm to enable such new services while eliminating the limitations of the Internet through network virtualization [13]. The key point in SDN is decoupling the network's control logic (the Control Plane) and forwarding logic (the Data Plane) [14]. After this separation, routers and switches become simple forwarding devices while the central controller becomes the brain of the network. The controller collects network state information, makes all the decisions (e.g., computes paths, allocates resources), and conveys these decisions to routers/switches through an application programming interface (API) such as OpenFlow [15]. Assuming that the cloud resources will be connected through an SDN-based network that can support per-flow routing decisions as in [3], we can now focus on how to compute bandwidth constrained paths for multiple flows.

Widest-Shortest algorithm [7]. If there is a path for f_k, we can subtract the bandwidth requirement r_k from the bandwidth of each link along that path. Then, we can repeat the process for the next flow f_{k+1} until we consider all the flows. Actually, this heuristic can perform well when the network resources are abundant. However, in the case of busy networks, this heuristic will often fail while there might be a better feasible solution. As a matter of fact, to find such a solution, the authors in [16] extended this iterative heuristic. In essence, they preempt the previously found paths and re-try the not-sent requests in several different orders. While this improves the amount of sent demands, it significantly increases the computation time.

Our goal here is to propose a new heuristic that can improve the performance in maximizing the amount of sent demands while significantly reducing the computation time when all the flows are between two cloud sites. After demonstrating the inadequacy of the pure iterative heuristic, we present the details of our proposed heuristic and its computation time analysis.

A. Failure of pure iterative heuristics

Consider the sample graph in Figure 1, where the available bandwidth of each link is given in parentheses. Suppose we

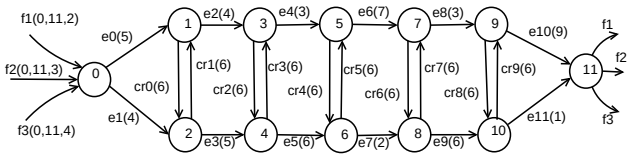

Fig. 1. A sample graph with 12 nodes and 22 links.

need to send three flows from node 0 to node 11 with the bandwidth demands of 2, 3, and 4. First, we will find the Widest-shortest path, which is shown by dashed links in Figure 2. The Widest path will start from v_0 then continue

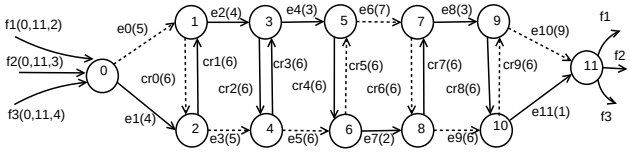

Fig. 2. Widest-Shortest Path.

with $v_1, v_2, v_4, v_6, v_5, v_7, v_8, v_{10}, v_9$ and end with v_{11}. The maximum bandwidth capacity for the chosen widest path will be 5. We can try to send the given three flows in any order. By inspection, one can easily see that after sending any one of these flows and reducing link bandwidths along that path, we will not have any more feasible paths for the other flows in the residual network. For example, if we send $f_1(0, 11, 2)$ and $f_2(0, 11, 3)$ on the widest path, then we cannot find any path for $f_3(0, 11, 4)$. Likewise, if we send $f_3(0, 11, 4)$ on the widest path, then there will not be any feasible path for the other flows.

Fortunately, again by inspection, one can see that we can actually satisfy the bandwidth requirements of all these three flows, if we carefully assign the flows to different links as shown in Figure 3, where flows that are assigned to each link are shown in brackets. Unfortunately, finding such an

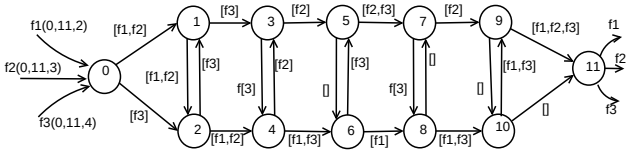

Fig. 3. Feasible assignments of the flows with our heuristic.

assignment is not an easy task as the problem is NP-Hard. So, our goal is to develop a new heuristic that can efficiently do such assignments by using various mechanisms.

B. Details of the proposed heuristic

The pseudo-code of our proposed heuristic is given in Algorithm 1. Our heuristic mainly consists of four stages: finding level cuts based on hop count information (Line 1), identifying the set of flows that can pass through the links on the level cut which has the minimum total bandwidth (Lines $2-10$), assigning the flows to the links on level cuts that are likely to be bottleneck (Lines $11-15$), and determining a complete path for each flow by finding and concatenating the paths between the assigned links on the consecutive level cuts (Lines $16-28$).

In the **first stage**, we need to determine level cuts. We can formally define a *level cut* as the set of the links that are at the same distance (minimum number of hops) from the source node s and whose removal will disconnect the source and destination. We can easily find such level cuts using the breath first search *(BFS)* algorithm, as presented in Algorithm 2. Basically, we start from the source node s and explore the graph level-by-level based on *BFS* and set the levels of the encountered links in a non-decreasing manner, as illustrated in Figure 4. If a link (u, v) is not encountered during the

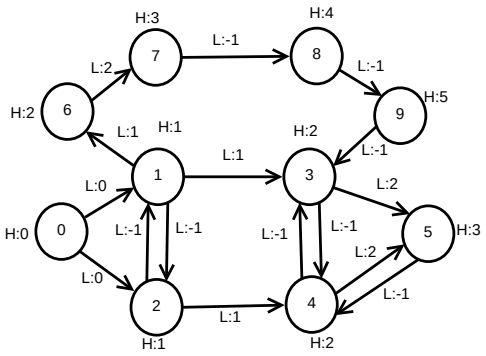

Fig. 4. Level determination.

search process, we keep its level as -1, indicating that it is not part of any level cut.

Intuitively, a level cut represents the set of the links that must be used to carry all the flows one step ahead towards the destination. So, we call the links in a level cut as *straight*

Algorithm 1 Proposed Level Cut algorithm

Input: Graph $G(N, A)$, $F = \{(s, d, r_k) : k = 1, 2, \ldots, K\}$, randomization factor β, and best fit factor γ
Output: P, a set of paths that maximize routed demand

Compute Levels($G(N, A)$, s, d)
$totalDemand = \sum_{k=1}^{K} r_k$
for $L = 0$ *to* $h[d]$ **do**
  $totalBW_L = \sum_{\{(i,j):level(i,j)\ is\ L\}} bw(i, j)$
end for
if $min(totalBW_L) < totalDemand$ **then**
  run randomized $0-1$ *knapsack* to find F', the new set of flows that can best fit into $min(totalBW_L)$
  $totalDemand = \sum_{k=1}^{K'} r_k$
  $F = F'$
end if
for $L = 0$ *to* $h[d]$ **do**
  if $(totalDemand \leq totalBW_L \leq \gamma * totalDemand$ **then**
    $BESTFIT(L, F, \beta)$
  end if
end for
for $f_k(s, d, r_k) \in F$ **do**
  $sequence_k = \{s\}$
  for $L = 0$ *to* $h[d]$ **do**
    if $Sharing_L$ is $true$ **then**
      Find the link (i, j) such that $level(i, j) = L$ and $x_{ij}^k = 1$
      $sequence_k = sequence_k \cup \{i, j\}$
    end if
  end for
  $sequence_k = sequence_k \cup \{d\}$
  Compute p_k by concatenating bandwidth-constrained shortest paths between the consecutive nodes in $sequence_k$, except for the nodes i and j when (i, j) is a link and has already been assigned to carry f_k.
  $P = P \cup p_k$
end for
return P

Algorithm 2 Compute Levels

Input: Graph $G(N, A)$, Source s, Destination d
Output: Levels

for $\forall\ (u, v)\ \in\ A$ **do**
  $level(u, v) = -1$
  $h[u] = \inf$
end for
$Q1 = \{s\}$ **and** $Q2 = \{\}$, where $Q1$ and $Q2$ are FIFO queues
$h[s] = 0$
while $d \notin Q1$ **do**
  while $Q1 \neq \phi$ **do**
    $u = dequeue(Q1)$
    for $\forall\ v\ \in\ AdjcentList(u)$ **do**
      if $h[u] \leq h[v]$ **then**
        $level(u, v) = h[u]$
        $h[v] = h[u] + 1$
        $enqueue(Q2, v)$
      end if
    end for
  end while
  $Q1 = Q2$
  $Q2 = \{\}$
end while

Algorithm 3 BESTFIT

Input: L, F, Randomization value β
Output: inDemand, outDemand, $Sharing_L$, x_{ij}^k

$sort(\{(i, j) : level(i, j)\ is\ L\})$ in decreasing order w.r.t. $bw(i, j)$
for β times **do**
  Randomly shuffle flows in F
  Assign each f_k to a feasible link $(i, j) \in \{(i, j) : level(i, j)\ is\ L\}$ and set x_{ij}^k to 1
  if $\forall f_k(s, d, r_k) \in F$ are assigned **then**
    $Sharing_L = true$
    Based on the fit, set $inDemand[i]$ and $outDemand[j]$ for $(i, j) \in \{(i, j) : level(i, j)\ is\ L\}$
  end if
end for
$Sharing_L = false$

links (denoted as by $e(bw)$ in the figures) as they are expected to carry the flows in one direction. We call the other links that are not in any level cuts as *cross* links (denoted by $cr(bw)$ in the figures) as they can be used to carry flows between or within level cuts in the final stage of our heuristic.

In the **second stage**, we first check if the sum of all requested demands ($totalDemand$) exceeds the total bandwidth ($totalBW$) of minimum level cut, which has the minimum $totalBW$ among all level cuts. If so, we rearrange the set of flows, so that we can get a new set of flows for which the $totalDemand$ is less than or equal to the $totalBW$ of the minimum level cut. For this, we simply use a modified knapsack algorithm with randomization.

In the **third stage**, we assign the flows to the links in each bottleneck level cut. To determine a bottleneck cut, we find the total bandwidth on each level cut. If $totalBW_L$ is greater than γ times $totalDemand$, then there is no need for an action here as the next stage can easily send all requests through that level cut. However, if $totalBW_L$ is less than γ times $totalDemand$, then this cut is considered to be a bottleneck cut and we need to carefully assign the flows to the links on that level cut. In order to share flows on links in a level cut, we use randomized best fit algorithm presented in Algorithm 3. We run the best fit algorithm β times.[2] When we share flows to each link (i, j), head node i keeps flows in $outDemand$ list and tail node j

[2]Note in our experiments we set γ to 1.5 and β to 3, as they provide the best trade-off between computation time and performance.

keeps flows in $inDemand$ list. We assume that there is an internal link between $inDemand$ and $outDemand$ with the bandwidth of ∞.

In the **fourth stage**, our heuristic first identifies the sequence of links in the level cuts that are assigned to carry each flow f_k. It then tries to find bandwidth-constrained shortest paths between the consecutive cuts and concatenates them to find a complete path for each flow. Let assume flow $f_k(0, 10, r_k)$ is shared on links (v_2, v_4), and (v_5, v_8) in a graph. So, $sequence_k$ has nodes $(v_0, v_2, v_4, v_5, v_8, v_{10})$ where v_0 is source node and v_{10} is destination node. Our heuristic searches shortest paths from v_0 to v_2, from v_4 to v_5 and from v_8 to v_{10} and concatenate them to the create complete path between v_0 and v_{10}. It does not search paths between $v_2 - v_4$ or $v_5 - v_8$, because the shared links, (v_2, v_4) and (v_5, v_8), are already assigned to carry that flow in previous stage. When finding shortest paths, our algorithm uses the Constraint Shortest Path First (CSPF) metric which is the reciprocal of residual bandwidth of links [19].

C. How the proposed heuristic works

To clarify the key ideas behind our heuristic, we will illustrate its execution using the sample graph in Figure 5. In the first stage, we found level cuts and we grouped links

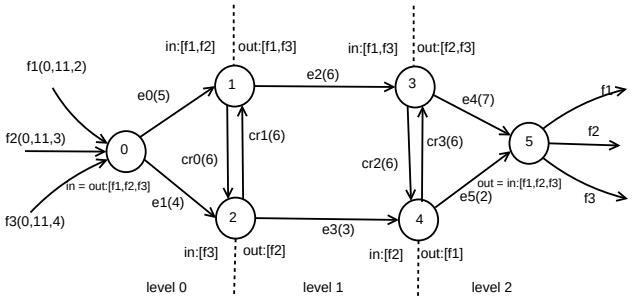

Fig. 5. Determining the level cuts and sharing demands (*in* and *out* represent *inDemand* and *outDemand* lists, respectively).

by their levels. In the second stage, we computed the total bandwidths of links in each level cuts. The total bandwidths of links in each levels are equal to 9. Since, none of the total bandwidths of level cuts is less than $totalDemand$, we proceeded to next stage without rearranging flow set. In the third stage, we applied the randomized best fit algorithm. Suppose it assigns f_1 to e_0, e_2, e_5; f_2 to e_0, e_3, e_4; and f_3 to e_1, e_2, e_4. Accordingly, the assigned flows put into inDemand and outDemand lists of the corresponding head and tail nodes of each link. In the fourth stage, we computed complete paths for $f_1(0, 5, 2)$, $f_2(0, 5, 3)$, and $f_3(0, 5, 4)$ by finding paths between level cuts. As an example, consider f_2 for which $sequence_2$ is $(v_0, v_1, v_2, v_4, v_3, v_5)$. We then searched paths from v_1 to v_2 and from v_4 to v_3. Finally, we concatenated these paths with already assigned links (v_0, v_1), (v_2, v_4), and (v_3, v_5) to create complete path for f_2.

D. Computational time analysis

Our heuristic has four stages. The first stage finds hop counts and determines level cuts of graph using BFS algorithm. So the time complexity of the first stage is $\mathcal{O}(n + m)$ where n is the number of nodes and m is the number of links in the graph. The second stage only arranges flows based on minimum cut using a randomized knapsack algorithm that shuffles flows β times to find best fit on total bandwidth of links in minimum cut and eliminates other flows which do not fit. If this case happens, rest of the algorithm uses this new flow set. The running time of this stage is $\mathcal{O}(\beta\big|F\big|)$, $\big|F\big|$ is number of flows. The third stage shares flows on links in each level cut if the total bandwidth of the links in that level cut is between total demand and γ times total demand. For sharing, we shuffle flows and try to assign them to links using best fit. We try this process at most β times for each level cut. So, the running time of the third stage is $\mathcal{O}(\beta\mathcal{L}\big|F\big|\big|m\big|)$, where $\mathcal{L}$ is number of level cuts (i.e., minimum number of hops between source and destination). Since β is constant, the running time of the second and third stages of the algorithm become $\mathcal{O}(\big|F\big|)$ and $\mathcal{O}(\mathcal{L}\big|F\big|\big|m\big|)$, respectively. Finally, the fourth stage basically uses a modified shortest path algorithm for each flow between the the consecutive bottleneck level cuts. Thus, the worst-case running time of this stage would be $\mathcal{O}(\big|F\big|\mathcal{L}(Dijkstra))$, which is also the overall running time of the proposed heuristic.

IV. Experimental Results

To evaluate our algorithm we implemented a simulator in Java. We run our simulator on a computer which has Intel Core(TM) $i7 - 4710HQ\ CPU\ @2.50\ GHz$ with $12GB$ RAM. We compare our algorithm against Greedy Algorithm with Preemption (GAP) in [16], which provides the highest performance in maximizing the total routed demand to the best of our knowledge. As the performance measure, we consider the total amount of sent flow and the execution time of each algorithm. We normalized the execution times based on our algorithm.

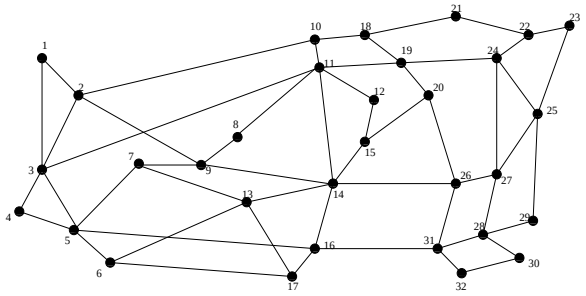

Fig. 6. Modified ANSNET topology.

For our tests, we first use a realistic topology by adding new links to ANSNET in [20], as seen Figure 6. We also consider random graphs generated by BRITE [21] with different densities (M) and randomly assigned bandwidths. Number of flows (K) is between $3-15$ with randomly assigned demands. When

generating graphs, link weights, and requested demands, we used common random numbers to minimize the variance and thus obtain a better confidence level on our comparison results. We run each algorithm 1000 times and report their averages. I all tests, we assign $\gamma = 1.5$, $\beta = 3$.

First Test

Our first test is based on ANSNET topology [20]. In each iteration, bandwidth capacity of each link is randomly generated from $uniform(10, 100)$. Then for each flow set, we randomly generate source s and destination d from $uniform(1, N)$, and r_k from $uniform(1, 40)$. As shown in Figures 7 and 8, our algorithm increases performance by $\%2 - 10$ in maximizing the total routed demand (or bandwidth) while significantly reducing the computation time. As the number of flows (K) increases, the computation time of GAP algorithm naturally increases significantly, as it preempts existing paths and retries many different possibilities.

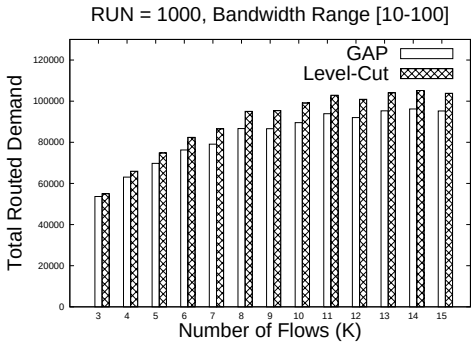

Fig. 7. First Test: Total Routed Demand.

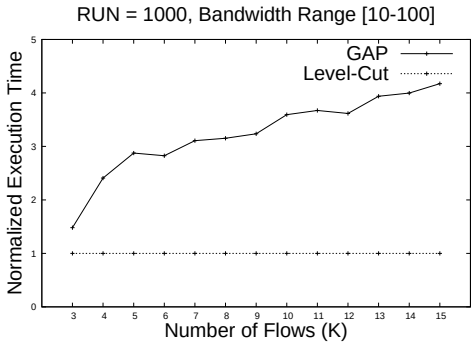

Fig. 8. First Test: Normalized Execution Time.

Second Test

Our second test is based on random graphs generated by BRITE [21]. We have tried different densities (M) and observed the similar trends. Due to page limitations, we report the results with $M = 5$. But more results can be found in our technical paper [22]. We generated 100 random graphs with density $M = 5$. Bandwidth capacity for each link is randomly generated from $uniform(10 - 100)$. We again consider $K = 3 - 15$ flows. For each flow set, we randomly generate source s and destination d from $uniform(1, N)$, and r_k from $uniform(1, 60)$ for all number of flows. As shown in Figures 9 and 10, both algorithms have similar performance in maximizing the total routed demand (our algorithm increases up to $\%4.5$) while ours again provides significant reduction in computation time as K increases.

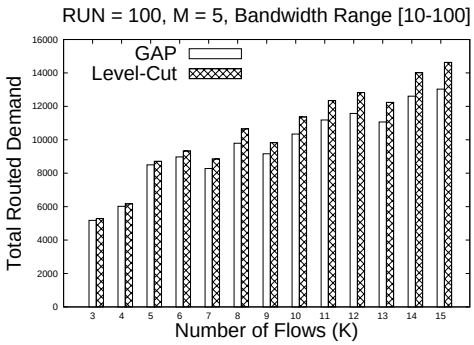

Fig. 9. Second Test: Total Routed Demand.

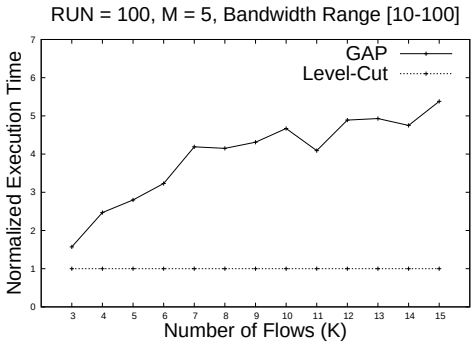

Fig. 10. Second Test: Normalized Execution Time.

Third Test

In our third test, we want to see how demand requirement range affects the performance. Here we generated bandwidth requirements r_k from $uniform(1, 10)$, $uniform(1, 20) \ldots uniform(1, 100)$, while keeping the topology, and the number of flows fixed. Again, we observed similar trends under different topologies and flow numbers as can bee seen in [22]. Thus we present one set of these results here based on randomly generated 100-node graphs. We fixed the number of flows (K) to 4 and density parameter M to 7 in random graphs. As before, we randomly generated bandwidth capacities from $uniform(10 - 100)$, source s and destination d from $uniform(1, N)$ for each flow set.

As shown in Figures 11 and 12, our algorithm again gets similar performance in maximizing the total routed demand, while providing significant reduction in computation time. One can see that total routed demand is increasing for both approaches until $uniform(1, 80)$. It then begins to show a declining tendency. This is because, bandwidth requirements of flows begin to exceed link capacities, which prevent to route some flows. Regarding computation time, we observe

that both algorithm has the similar computation times when we deal with small demands. However, as the demand ranges increase, again the computation time of GAP algorithm increases significantly as it ends up re-trying many possibilities to achieve a good performance in maximizing routed demands.

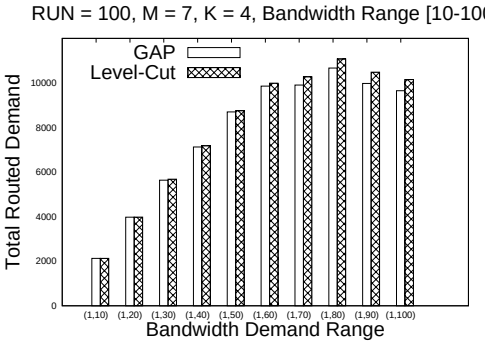

Fig. 11. Third Test: Total Routed Demand.

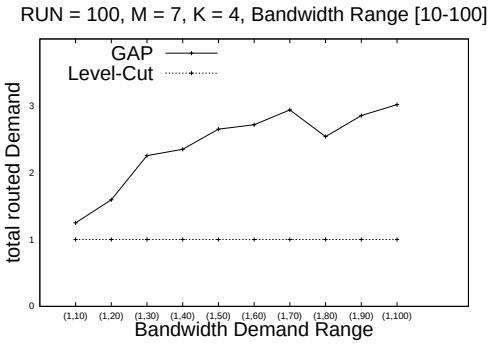

Fig. 12. Third Test: Normalized Execution Time.

V. Conclusions and Future Works

We have focused on a special case of the Unsplittable Flow Problem (UFP), where the multiple flows with different bandwidth constraints are between two cloud sites. Since this special case is still NP-Hard, we proposed a heuristic algorithm. Our heuristic algorithm first determines level cuts between the source and destination. After rearranging flow set based on total bandwidth of minimum cut, it then assigns the flows to the links in the bottleneck cuts using randomized best fit approach. It finally tries to forward flows between the bottleneck cuts using a bandwidth constrained shortest path per flow. We compared our algorithm against an effective heuristic provided in [16]. We demonstrated that our algorithm achieved similar or better performance in maximizing total routed demand while significantly reducing computation time. So, our solution will be much more useful in real-time path computation scenarios.

As the future work, we plan to generalize our algorithm to find paths for multiple flows among different source-destination pairs. In addition, we will study how to deal with inaccurate state information while the current version assumes accurate state information.

References

[1] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, p. 13, 2009.

[2] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.

[3] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[4] Q. Ma and P. Steenkiste, "On path selection for traffic with bandwidth guarantees," in *Proceedings of the IEEE International Conference on Network Protocols (ICNP '97)*, 1997, pp. 191 –202.

[5] A. Orda, "Routing with end-to-end QoS guarantees in broadband networks," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 365–374, 1999.

[6] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[7] R. A. Guérin and A. Orda, "Qos routing in networks with inaccurate information: theory and algorithms," *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 3, pp. 350–364, 1999.

[8] M. M. Atanak, A. Dogan, and M. Bayram, "Modeling and resource scheduling of real-time unsplittable data transfers," *Appl. Math*, vol. 9, no. 2, pp. 1067–1080, 2015.

[9] B. Ma and L. Wang, "On the inapproximability of disjoint paths and minimum steiner forest with bandwidth constraints," *Journal of Computer and System Sciences*, vol. 60, no. 1, pp. 1–12, 2000.

[10] N. Garg, V. V. Vazirani, and M. Yannakakis, "Primal-dual approximation algorithms for integral flow and multicut in trees," *Algorithmica*, vol. 18, no. 1, pp. 3–20, 1997.

[11] J. M. Kleinberg, "Approximation algorithms for disjoint paths problems," Ph.D. dissertation, Citeseer, 1996.

[12] J. Moy, "OSPF version 2," IETF, Standards Track RFC 2328, April 1998.

[13] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[14] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[16] K. Walkowiak, "New algorithms for the unsplittable flow problem," in *Computational Science and Its Applications-ICCSA 2006*. Springer, 2006, pp. 1101–1110.

[17] M.-C. Yuen, W. Jia, and C.-C. Cheung, "Simple mathematical modeling of efficient path selection for qos routing in load balancing," in *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*, vol. 1. IEEE, 2004, pp. 217–220.

[18] P. Kolman and C. Scheideler, "Improved bounds for the unsplittable flow problem," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2002, pp. 184–193.

[19] E. Crawley, H. Sandick, R. Nair, and B. Rajagopalan, "A framework for qos-based routing in the internet," 1998.

[20] D. E. Comer, *Internetworking with TCP/IP*, 3rd ed. Prentice Hall, Inc., 1995, vol. I.

[21] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: An approach to universal topology generation," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*. IEEE, 2001, pp. 346–353.

[22] E. Akin and T. Korkmaz, "Routing multiple unsplittable flows between two cloud sites with QoS guarantees," March 2016. [Online]. Available: http://www.cs.utsa.edu/∼korkmaz/research/levelcut