An Efficient Multipath Routing Algorithm for Multipath TCP in Software-Defined Networks

Jang-Ping Sheu, Lee-Wei Liu, Jagadeesha RB, and Yeh-Cheng Chang
Department of Communications Engineering, National Tsing Hua University
Hsinchu, 30013, Taiwan
sheujp@cs.nthu.edu.tw, gary80518@yahoo.com.tw, jagadeesha.rb@gmail.com, jas1123kimo@gmail.com

***Abstract*—Due to the rapid growth of Software-Defined Networks, we can manage network traffic easier than before. One of the major issues on SDN is the multipath routing protocols due to its complexity of optimum path selection among the multiple routes. In this paper, we focus on Multipath TCP (MPTCP). MPTCP is an extension of TCP that increases the throughput of TCP communication significantly by utilizing multiple paths transmission rather than a single path. We consider the multipath routing problem as a *k* Max-Min bandwidth disjoint paths one. The problem is to find *k* disjoint paths with relative higher throughput and the smallest bottleneck bandwidth of the *k* paths is the maximum. Since this problem is NP-complete, we propose a heuristic algorithm to solve this problem in polynomial time. The simulation results show that our proposed algorithm perform better than previous work in terms of average throughput and average hop count.**

Keywords—Max-min disjoint paths, multipath routing algorithm, multipath TCP, software-defined networks

I. INTRODUCTION

Software-Defined Networks (SDN) [1] is a new network paradigm. SDN is divided into three layers; namely, control plane layer, data plane layer, and application layer. The SDN control plane makes the decision for network flow and uses standardized interface to communicate with SDN data plane. OpenFlow, [2] is an open standard interface for SDN control plane to communicate with SDN data plane. SDN data plane employs programmable OpenFlow switch. SDN applications exist in the application and communicate with SDN control plane via northbound API. There are many varieties of applications such as routing, traffic engineering, multicasting, security, access control, bandwidth management, quality of service, and energy usage. In our research, we use SDN mainly to exploit its feature of dynamic path selection.

One of the major research topic on SDN is multipath routing. In [3] [4], the authors showed the benefits of the multipath routing and use of disjoint paths in OpenFlow. Here, we focus on the Multipath TCP (MPTCP), a new coming transport layer protocol, on SDN. MPTCP [5][9][11] is an extension of TCP. The MPTCP sender divides a regular TCP flow to several TCP sub-flows. The MPTCP receiver aggregates sub-flows and reassembles the packet which comes from different paths. When a sub-flow fails, there are other sub-flows exist such that the connection will not fail. This improves the robustness of the TCP flow connection. The MPTCP mainly relies on Open Shortest Path First/Equal Cost Multipath (OSPF/ECMP) to hash different sub-flows to different paths. It allows us to use multipath for a single connection.

However, the authors in [6] indicated that OSPF/ECMP routing is not flexible enough if all the sub-flows of a connection use the same path; it leads every sub-flow to compete for the same resources that will reduce the network throughput. To solve this problem, we can use *k* disjoint paths to route the sub-flows of an MPTPC connection. In [7], the authors showed how path selection for a sub-flow affects network throughput. In their analysis, many of the performance benefits are the results of spreading the load over many paths. In addition, the authors in [8] addressed the problem of head-of line blocking in the network. In MPTCP, this phenomenon is caused by the packets that are scheduled on the low-delay sub-flow, which have to wait for the high-delay sub-flow's packets to arrive in the out-of-order queue of the receiver. To solve this problem, we try to maximize the minimum bottleneck bandwidth path (least bandwidth link) of the *k* disjoint paths that are a *k* Max-Min bandwidth disjoint paths.

The authors in [10] further proved that four versions of the problem (i.e., the graph is directed or undirected and the paths are edge-disjoint or vertex-disjoint) are strongly NP-hard even *k* equals to two. It costs exponential time to find an optimal solution for the above problems. Therefore, some researchers [12] have proposed various approximation algorithms for the special cases of *k* Max-Min disjoint paths problem, but cannot be used on directed weighted graph. The authors in [13] focused on reducing the time complexity for optimal solution by formulating the problem to find the first *k* disjoint paths with the largest bottleneck bandwidth (*k*-Max disjoint paths) on the graph, instead of *k* Max-Min disjoint paths. Our intuition for better performance of throughput is due to the multiple path selection instead of a single one by using MPTCP.

To maximize the minimum bottleneck bandwidth path of *k* disjoint paths, our proposed algorithm is divided into two phases. The first phase is done by modifying Dijkstra's shortest path algorithm to create the candidate paths of *k* disjoint paths. The second phase uses greedy scheme to select *k* disjoint paths to maximize the minimum bottleneck bandwidth of the *k* disjoint paths. We implement our algorithm on Ryu controller and use Mininet as the SDN data plane. Waxman random network and

Internet2 deployed in U.S and Cernet deployed in China are our simulation topologies. The simulation results show an improvement in the average throughput per MPTCP flow by more than 10% compared with the first *k*-Max disjoint paths. This result indicates that our algorithm has better throughput than previous work.

The rest of the paper is organized as follows. In section II, we describe our algorithm in detail. Simulation results are shown in section III. Conclusion is given in section IV.

II. *K* MAX-MIN DISJOINT PATHS ALGORTIHM

Our algorithm includes two phases. In the first phase, we modify the Dijkstra's shortest path algorithm to find a set of candidate paths between a pair of source and destination. In the second phase, we use the greedy technique to pick the *k* Max-Min bandwidth disjoint paths from the candidate paths set.

In an SDN network, it consists of SDN-enabled switches, SDN controller, and hosts. We consider the SDN network as a weighted graph $G = (V, E)$ in which V is a set of vertices and E is a set of edges interconnected vertices in V. Each vertex in V represents a switch in SDN and each edge in E represents a switch link in SDN. Every link is bidirectional. For each edge $(u, v) \in E$, $rb(u, v)$ denotes the remaining bandwidth of the edge (u, v). Let $s \in V$ be a source node and $t \in V$ be a destination node. Assume there is a path from s to t, the minimum remaining bandwidth on the edge of a path is called *bottleneck bandwidth* of the path. If there are multiple paths from s to a node v, the maximum bottleneck bandwidth (MBB) of node v is the bottleneck bandwidth of a path, which is maximum among all the paths. The minimum hop count (MHC) from s to v is the number of hops of the shortest path from s to v. Let *v.mbb* and *v.mhc* denote the MBB and MHC from source s to node v, respectively. In Fig. 2.1, the alphabet in each circle represents a switch ID, and the two numbers in each circle represent the *v.mbb* and *v.mhc* of node v, respectively. The edge weight represents the remaining bandwidth between two switches. For example, in Fig. 2.1 the bottleneck bandwidth of path $s - a - d - t$ is 8. The *t.mbb* and *t.mhc* are 8 and 3, respectively.

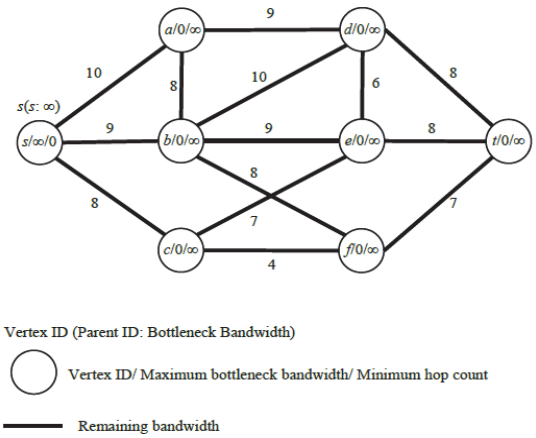


Figure 2.1: An example of SDN with eight switches.

A. *Find a set of candidate paths*

Here, we explain the main steps of the phase one algorithm. Initially, $v.mbb = 0$ and $v.mhc = \infty$ for all $v \in V$ except s. The *s.mbb* and *s.mhc* are set to ∞ and 0, respectively as shown in Fig. 2.1. All nodes are marked as unvisited and undiscovered at the beginning of this phase. First, we visit from source s and step by step to nodes t. If we visit a node v, we relax all its outgoing edges connected to unvisited neighbors. For example, a, b, and c are the nodes s relax at the beginning. After an edge (u, v) is relaxed, node v is denoted as discovered. When there are multiple unvisited neighbors, the node with the least MHC is discovered first. This is because we want to get more candidate paths in this way. If multiple nodes have the same MHC, we break the tie in random selection. In this example, as in Fig. 2.1, since $a.mhc = b.mhc = c.mhc$, and their MBB is smaller than s and are relaxed by s. Therefore, we discover the three nodes a, b, and c in random order, here node a is the random selection. Thus, the weight of node a should be relaxed to 10 and node s is set to be visited.

Note that, we have two types of relax operations depends on the two nodes attached on an edge. To relax an edge (u, v), if $u.mhc \leq v.mhc$, we execute one-way relax operation. Otherwise, we execute two-way relax operation. The relax operation is to find the MBB and MHC of a node v from source s. In addition, we also need to record the parents of node v and the bottleneck bandwidth of the paths from node v's parents. The two-way relax operation is to relax edge (u, v) and edge (v, u) simultaneously. The two-way relax operation tries to find more candidate paths other than the shortest paths between source and destination. The one-way relax operation for edge (u, v) is described as follows. If $v.mbb < min(u.mbb, rb(u, v))$, the *v.mbb* is updated to $min(u.mbb, rb(u, v))$. If $v.mhc > u.mhc + 1$, the *v.mhc* is updated to $u.mhc + 1$. After the one-way relax operation, node u is denoted as the parent of node v. Since there may exist more than one path from source s to node u, the node v need to keep the bottleneck bandwidth of all the paths from node u. To reduce the time complexity of our algorithm, the maximum number of paths from a parent node to a node is restricted to m. The value m depends on the computation capacity of an SDN controller. Bigger the m is, we have the better k Max-Min disjoint paths but spends a more computational time to find the k Max-Min disjoint paths. Let $v(u: v_1, v_2, \ldots, v_h)$ denote the h bottleneck bandwidth values of h paths from source s to node v through one of its parents u, where $h \leq m$. Note that, a node can have multiple parents if the node's incoming edges was relaxed by different nodes. Assume u has x paths from s to u through all u's parents and their bottleneck bandwidth are denoted as bb_1, bb_2, …, and bb_x in decreasing order. When u relax edge (u, v), $v_i = min(bb_i, rb(u, v))$, for $1 \leq i \leq min(m, x)$.

For example, in Fig. 2.2 and the following examples we assume $m = 4$. Since $s.mhc \leq a.mhc$, we execute one-way relax operation. Thus, *a.mbb* and *a.mhc* are updated to 10 and 1, respectively. And $a(s: 10)$ represents that there exists a path with bottleneck bandwidth 10 from source s to node a through a's

parent node *s which is maximum among a,b,c* . Similarly, after nodes *b* and *c* are relaxed, we have *b.mbb* = 9, *b.mhc* = 1, *b*(*s*: 9), *c.mbb* = 8, *c.mhc* = 1, and *c*(*s*: 8) as shown in Fig. 2.2. After all the unvisited neighbors of a node are discovered, we mark the node as visited. Next, we will visit the discovered node which has the largest MBB. When there are multiple discovered nodes with the same MBB, we visit the node with smaller MHC first. If there are multiple discovered nodes with the same MBB and MHC, we randomly select one of them.

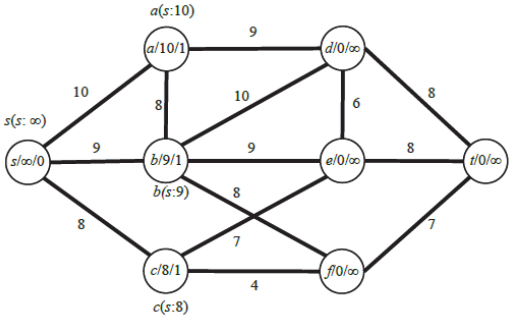
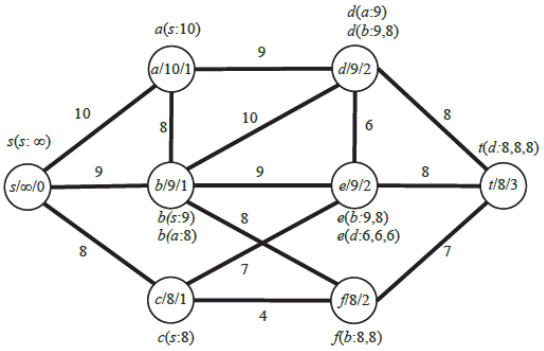


Figure 2.2: Source node *s* relaxes edges (*s*, *a*), (*s*, *b*), and (*s*, *c*).

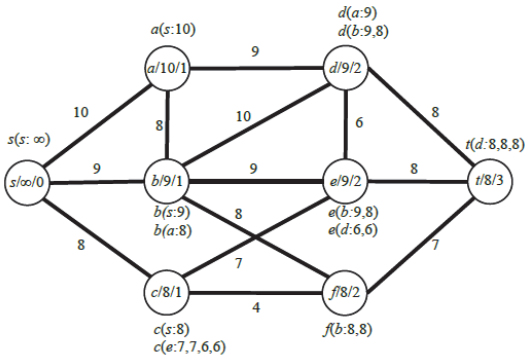
In the following, we use an example to illustrate the two-way relax operation. Fig. 2.3(a) shows the results of nodes *a*, *b*, and *d* are visited as *b* and *d* are neighbors of *a,* visited earlier. The nodes *b* and *d* have the largest maximum bottleneck bandwidth, we visit node *b* and *d,* in addition, both have same minimum hop count we visit them randomly. Thus, the unvisited discovery nodes are nodes *c*, *e*, *f*, and *t*. Note that, destination *t* is always the last one to be visited because we prefer to get more candidate paths other than the shortest path from source *s* to destination *t*. Since *e.mbb* > *c.mbb* = *f.mbb.* The node *e* will relax edges (*e*, *c*) and (*e*, *t*). Since *e.mhc* > *c.mhc*, we need to perform two-way relax operation. The two-way relax operation will relax edges (*e*, *c*) and (*c*, *e*) simultaneously. To relax edge (*e*, *c*), we only select the first four paths with the larger bottleneck bandwidth from node *e* to be the candidate paths of node *c* since *m* = 4. So, we have *c*(*e*: 7, 7, 6, 6) after relaxed the edge (*e*, *c*) as shown in Fig. 2.3(b). Since the fifth path with bottleneck bandwidth = 6 in node *e* is not used for node *c*, we delete this path from node *e*. The *c.mbb* and *c.mhc* remain unchanged according to the principle of one-way relax operation. To relax edge (*c*, *e*), since there is only one path through node *c*, the node *e* gets a new parent *c* which is denoted *e*(*c*: 7). After that, we visit nodes *c*, *f*, and *t*. When destination *t* is visited, all the candidate paths from source *s* to *t* are determined. Fig. 2.4 shows the final result of phase one.

B. Select k Max-Min Bandwidth Disjoint Paths

Once we obtain a set of candidate paths, we use the greedy technique to select *k* disjoint paths from the set of candidate paths and try to maximize the minimum bottleneck bandwidth path of the *k* disjoint paths. Let $p_1, p_2, \ldots, p_k$ denote the *k* disjoint paths from *s* to *t* which are determined by phase two algorithm. First, we assume there are *x* candidate paths in destination *t*. Here, we can trace back the parents of *t* to find each candidate path from *s* to *t* and its corresponding bottleneck bandwidth.



(a): Nodes *a*, *b* and *d* relax their outgoing edges.



(b): Node *e* relaxes edge (*e*, *c*)

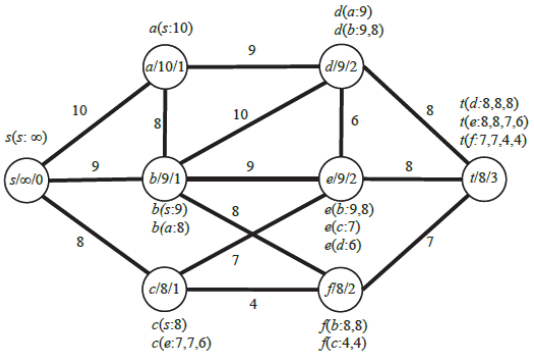


Figure 2.4: Final result of the phase one algorithm.

We sort all the candidate paths in decreasing order based on their bottleneck bandwidths and denote them as $cp_1, cp_2, \ldots, cp_x$. Let $b(cp_i)$ denote the bottleneck bandwidth of path cp_i, for $1 \leq i \leq x$. For example, in Fig. 2.5 $cp_1 = s - a - d - t$ and $b(cp_1) = 8$. Let table *T_disj*[][] denote a two-dimensional array stores the disjoint-path relations of the *x* candidate paths. If a path cp_i is disjoint with a path cp_j, the element *T_disj* [*i*][*j*] = *T_disj* [*j*][*i*] = true; otherwise *T_disj* [*i*][*j*] = *T_disj* [*j*][*i*] = false.

The phase two algorithm consists of *x* iterations. In each iteration, we will find a set of *k* disjoint paths. The best solution of the *x* iteration is our final result. Let *P_tmp* be a temporary array which is used to store the *k* disjoint paths in each iteration

i, for $1 \le i \le x$. In each iteration i, let $p_1 = cp_i$ and $P\_tmp[1] = cp_i$. Then we can find p_2 by checking the ith column of table T_disj [][] from T_disj [i][1] to T_disj [i][x] to find the first element = true. If the jth element is equal to true, it represents $p_2 = cp_j$ and let $P\_tmp[2] = cp_j$. If there are y disjoint paths in P_tmp, we can find the y+1th disjoint path by checking y columns of table T_disj[][] from the first row to the last row to find if there exists a row for all the y columns are true. If it exists, $P\_tmp[y+1] = cp_j$. Otherwise, we will go to next iteration. Note that, when we want to find a disjoint path other than the paths in P_tmp, we always check the candidate paths from the largest bottleneck bandwidth to the smallest one. Finally, we can find a set of k disjoint paths in each iteration. After x iterations, we have x sets of the k disjoint paths. We will select the one whose minimum bottleneck bandwidth of the k disjoint paths is maximum.

For example, let k = 3. In the first iteration of Fig. 2.5, we have $P\_tmp[1] = cp_1 = s - a - d - t$ and $b(p_1) = 8$. To find p_2, we can scan the first column of table T_disj[][] to find $p_2 = cp_4 = s - b - e - t$ and $b(p_2) = 8$. We have $P\_tmp[2] = cp_4$. Then, we scan the first and fourth columns of table T_disj[][] to find the third disjoint path $p_3 = cp_{10} = s - c - f - t$ and $b(p_3) = 4$ which is path disjoint with cp_1 and cp_4. Finally, we have P_tmp = {cp_1, cp_4, cp_{10}} and the minimum bottleneck bandwidth of the three disjoint paths is four. In the second iteration, let $P\_tmp[1] = cp_2$. We can repeat the above steps to find another set of three disjoint paths. After 11 iterations, we have 11 sets of three disjoint paths in Fig. 2.5. The best result of the three disjoint paths is $p_1 = cp_6 = s - c - e - t$, $b(p_1) = 7$, $p_2 = cp_1 = s - a - d - t$, $b(p_2) = 8$, $p_3 = cp_7 = s - b - f - t$, and $b(p_3) = 7$ as shown in Fig. 2.6. The minimum bottleneck bandwidth of the three disjoint paths is 7. To save the memory space, if the output result of the current iteration is better than the previous iteration, we will store the result of the current iteration; otherwise the current iteration result is abandoned. If we cannot find k disjoint paths in an iteration, we will return the maximum number of disjoint paths.

C. *Time Complexity Analysis*

In the first phase of our algorithm, we need to visit all nodes in graph $G(V, E)$. We implement a maximum heap to store every node in order to pop out the node which has the maximum

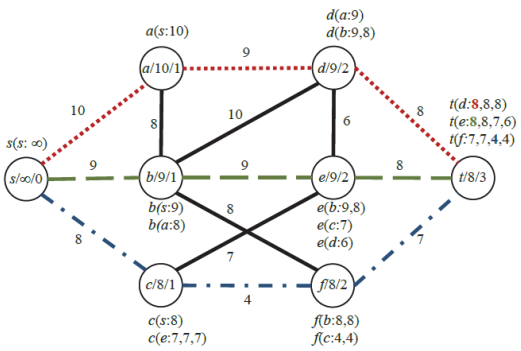


Figure 2.5: The three disjoint paths from s to t for P_tmp = {cp_1, cp_4, cp_{10}}.

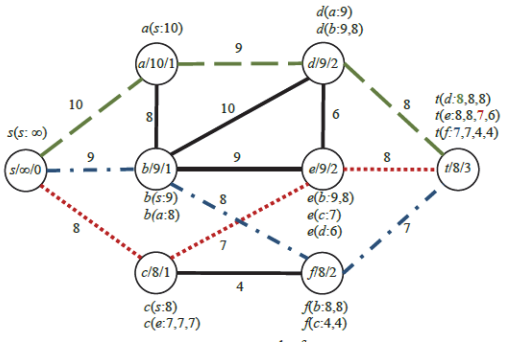


Figure 2.6: The best result of three disjoint paths for $p_1 = cp_6$, $p_2 = cp_1$, and $p_3 = cp_7$.

bottleneck bandwidth and it consumes $O(|V|\log|V|)$ for visiting $|V|$ nodes. Furthermore, we implement another maximum heap to store the bottleneck bandwidth of each path in each node. We pop out the largest m bottleneck bandwidths when we relax an edge and it consumes $O(\log|V|)$. There are $|E|$ edges in a graph. Therefore, the first phase of our algorithm runs in $O(|E|\log|V|) + |V|\log|V|)$.

In the second phase of our algorithm, we obtain a path in $O(|V|)$ by trace back the parent nodes from destination t. If the destination node t has p parents. Then, we have $x = mp$ candidate paths and it costs $O(x|V|)$ to get all the candidate paths. To check if two candidate paths are a disjoint path or not needs $O(|V|^2)$. Thus, we need $O(x(x + 1)/2|V|^2)$ to complete the table T_disj. Next, we need $O(kx)$ to find a set of disjoint paths beginning from each cp_i, for $1 \le i \le x$. Therefore, it costs $O(kx^2)$ to find x sets of disjoint paths. Finally, the time complexity (considering $x = mp$ is constant) of our algorithm is $O(x^2|V|^2 + kx^2)$. Thus, the time complexity of phase two is $O(|V|^2)$ and the time complexity of our algorithm is $O(|E|\log|V|+ |V|^2)$.

III. PERFORMANCE EVALUATION

In this section, we compare the performance of shortest path first (SPF), the first k-Max bandwidth disjoint paths algorithm [13], and our proposed algorithm on Waxman random graph, Internet2 and Cernet. We modify Linux kernel to install MPTCP and Mininet such that Mininet can simulate MPTCP traffic in an SDN environment. The Mininet creates a realistic virtual network, running the real kernel, switches and application code, on a single machine (VM, cloud or native). This makes us customize our SDN environment. We use Ryu as SDN controller to manage all the flow control and run OpenFlow 1.3 in our simulations. We design our proposed algorithm as an SDN application and decides all the paths for MPTCP traffics. When an MPTCP sender generates a sub-flow, the SDN controller can decide a path for the sub-flow and set the rule on the SDN switches along the path.

We use random topology and real SDN topology in our simulations. Waxman random graph is commonly used in network simulations for random topology. The link between two nodes is decided by the distance between two nodes on the graph. The longer the distance between two nodes, lowers the

possibility of the link exists. We generate topology with 10, 20, 30, 40, and 50 nodes and the link degree of each node is around five. Besides the random graph, we also evaluate our algorithm on the layer 2 of Internet2 and Cernet. By these two SDN topologies, we can evaluate our algorithm for more realistic scenarios.

At the beginning of our simulations, we start with zero traffic, and later generate random traffic. Then, we randomly pick a communication pair to generate an MPTCP flow and record the bottleneck bandwidth of each sub-flow using Iperf as our throughput testing tool. We test an end-to-end connection for 20 seconds and evaluate the throughput difference between the maximum and minimum bottleneck bandwidth paths. The average hop count of k disjoint paths is analyzed as well. In the following simulations, the parameter m used in our algorithm is set to 7. The maximum number of sub-flow k is set to 3. We also make k as a variable to analyze how k affects the result of our proposed algorithm and others.

In Fig. 3.1, our proposed algorithm has the largest throughput with the minimum bottleneck bandwidth path, which is better than the first k-Max disjoint paths algorithm and SPF. Besides, our proposed algorithm has a smaller throughput of the path with maximum bottleneck bandwidth than that of k-Max bandwidth disjoint paths algorithm because we increase the throughput of the path with minimum bottleneck bandwidth by sacrificing the throughput of the path with maximum bottleneck bandwidth. The average bottleneck bandwidth of k disjoint paths of our proposed algorithm is similar to the k-Max bandwidth disjoint paths algorithm. However, since we aim to solve k Max-Min disjoint paths, our proposed algorithm has smaller throughput difference between the paths with the maximum and minimum bottleneck bandwidths, which is 35% better than the k-Max bandwidth, disjoint paths algorithm. Thus, our scheme can reduce the impact of head-of-line blocking problem. Thus, we get a better average throughput of an MPTCP flow as in Fig. 3.2.

Therefore, Fig. 3.2 shows that our proposed algorithm has 8% better average throughput of an MPTCP flow than the first k-Max disjoint paths algorithm. Since SPF do not use the disjoint paths policy, the throughput of our scheme is 90% better than SPF. As the number of nodes increase, there exist more disjoint paths between a pair of end-to-end communication. Therefore, the average throughput of an MPTCP flow increases as the number of nodes increases.

In Fig. 3.3, we use Internet2 and Cernet as our network topologies. Since these two topologies have fewer links than the Waxman random graph, i.e. less disjoint paths between a pair of communication nodes, our proposed algorithm has higher average throughput of k disjoint paths than SPF and the first k-Max disjoint paths algorithm. Moreover, our proposed algorithm presents smaller throughput difference between minimum and maximum bottleneck bandwidth than the first k- Max disjoint paths and SPF.

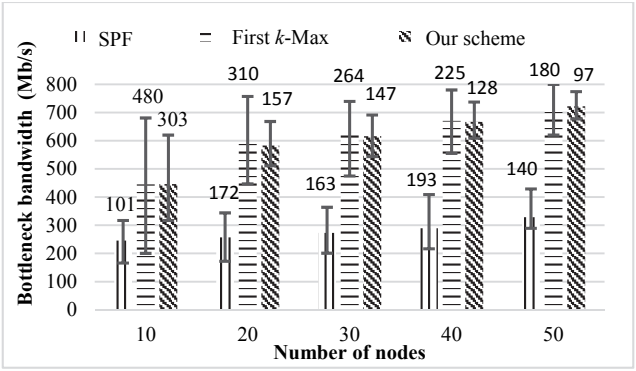


Figure 3.1: The throughput of maximum, average and minimum bottleneck bandwidth path in Waxman random graph

Therefore, Fig. 3.4 shows that the average throughput of our proposed algorithm is 15% and 33% better than the first k-Max disjoint paths and SPF, respectively.

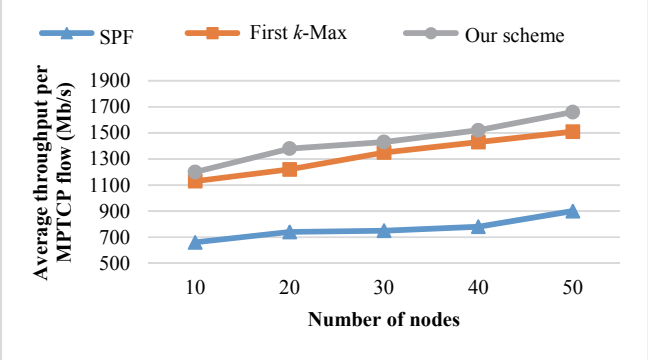


Figure 3.2: The average throughput of an MPTCP flow in Waxman random graph

Here, we make the number of sub-flow k as a variable on the Waxman random topology with 40 nodes. Figures 3.5 shows that our proposed algorithm has smaller throughput difference between the paths with the maximum and minimum bottleneck bandwidths on various k than that of the k-Max bandwidth disjoint paths.

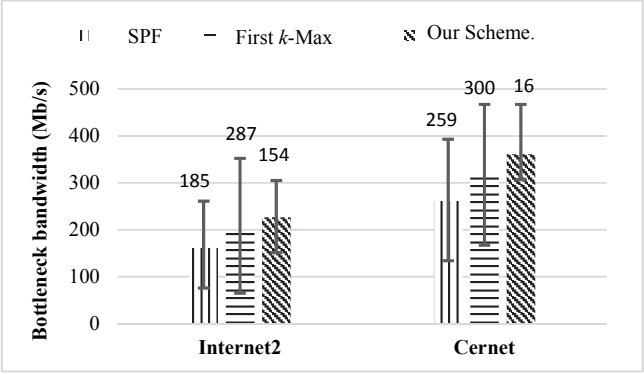


Figure 3.3: The throughput of maximum, average and minimum bottleneck bandwidth path in realistic SDN topology

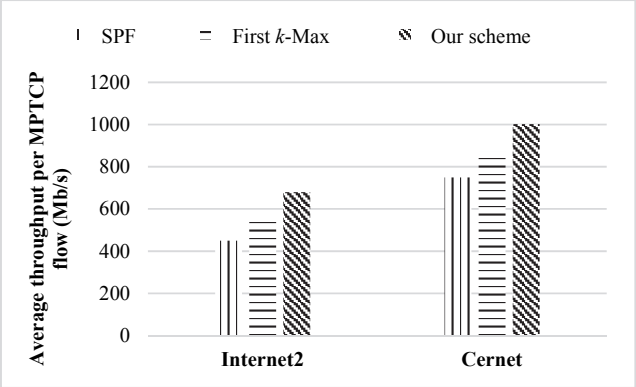


Figure 3.4: The average throughput of an MPTCP flow in realistic SDN topology

In addition, as the k increases, the average bottleneck bandwidth path of k disjoint paths decrease for both algorithms due to the smaller bottleneck bandwidth disjoint path is found. Since SPF always uses the shortest paths for end-to-end communication, the average bottleneck bandwidth path of k disjoint paths keep almost the same for various values of k.

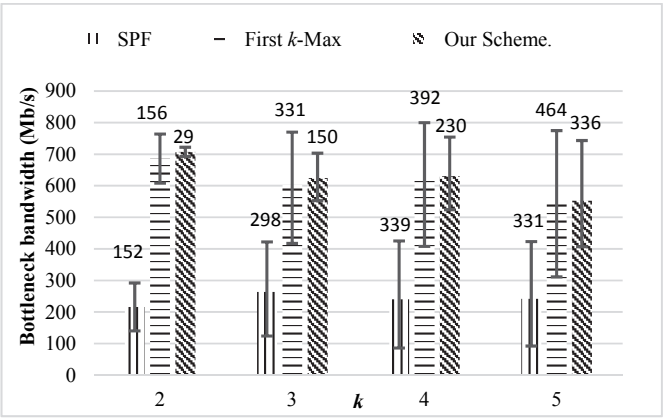


Figure 3.5: The throughput of maximum, average and minimum bottleneck bandwidth path with variable k

In Fig. 3.6, the average throughput of an MPTCP flow is increased as k increases.As the k increases, the gap of the average throughput of an MPTCP flow between our proposed algorithm and other two algorithms increases.

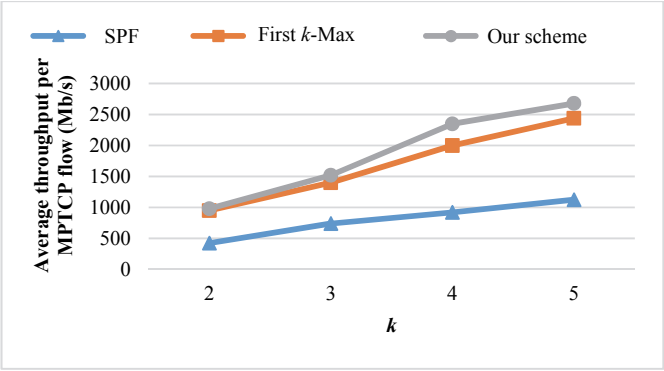


Figure 3.6: The average throughput of a MPTCP flow with various values of k

IV. Conclusion

In this paper, we proposed an efficient heuristic algorithm for k Max-Min bandwidth disjoint paths to improve the performance of an MPTCP flow, which can solve the disjoint paths problem and the head-of-line blocking problem. The proposed heuristic algorithm is computationally more efficient for network routing than the optimal solution. Our algorithm first finds a set of candidate paths from source to destination in polynomial time. Then, we use the greedy technique to choose k disjoint paths of high bottleneck bandwidth from the candidate paths. This results in high throughput improvement of an MPTCP flow. The simulation results show that the average throughput of an MPTCP flow of our proposed algorithm is better than the first k-Max disjoint paths and SPF.

References

[1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *Proceedings* of *ACM SIGCOMM,* pp. 1-12, New York, USA, August 2007.

[2] 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, April 2008

[3] M. Koerner, and O. Kao, "Evaluating SDN based Rack-to-Rack Multi-Path Switching for Data-Center Networks," *Procedia Computer Science*, Vol. 34, pp. 118-125, 2014

[4] R. van der Pol, M. Bredel, and A. Barczyk, "Experiences with MPTCP in an Intercontinental Multipathed OpenFlow Network," *Proceedings of the 29th Trans European Research and Education Networking Conference*, Maastricht, Netherland, June 2013.

[5] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, RFC 6182, "Architectural Guidelines for Multipath TCP Development," 2011.

[6] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A Roadmap for Traffic Engineering in SDN-OpenFlow Networks," *Computer Networks*, Vol. 71, No. 4, pp. 1-30, October 2014.

[7] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," *Proceedings of ACM SIGCOMM*, pp. 266-277, New York, USA, August 2011.

[8] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental Evaluation of Multipath TCP Schedulers," *Proceedings of ACM SIGCOMM Workshop on Capacity Sharing*, New York, USA, August 2014.

[9] Jingpu Duan, Zhi Wang, Chuan Wu,"Responsive Multipath TCP in SDN-based Datacenters ," *IEEE ICC international conference on communications*, pp. 6914-6919 , London, June 2015.

[10] C-L Li, S. T. McCormic, and D. Simchi-Levi , " The Complexity of Finding Two Disjoint Paths with Min-Max Objective Function ," *Discrete Applied Mathematics*, Vol. 26, No. 1, pp. 105-115, January 1990.

[11] Benjamin Hesmans,Hoang Tran-Viet,Ramin Sadre,d Olivier Bonaventure, " A First Look at Real Multipath TCP Traffic ," *Proceedings of 7th international workshop TMA* , pp. 233-245, 2015.

[12] P. Zhang, and W. Zhao, "On the Complexity and Approximation of the Min-Sum and Min-Max Disjoint Paths Problems," *Proceedings of Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pp. 70-81, Hangzhou, China, 2007.

[13] R. C. Loh, S. Soh, M. Lazarescu, and S. Rai, "A Greedy Technique for Finding the Most Reliable Edge-Disjoint-Path-Set in a Network," *Proceedings of 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 216-223, Taipei, December 2008.