Efficient Recovery Path Computation for Fast Reroute in Large-scale Software Defined Networks

Kun Qiu, Jin Zhao, Xin Wang, Xiaoming Fu, Stefano Secci

▶ **To cite this version:**

Kun Qiu, Jin Zhao, Xin Wang, Xiaoming Fu, Stefano Secci. Efficient Recovery Path Computation for Fast Reroute in Large-scale Software Defined Networks. IEEE Journal on Selected Areas in Communications, 2019, 37 (8), pp.1755-1768. 10.1109/JSAC.2019.2927098 . hal-02181090

HAL Id: hal-02181090
https://hal.science/hal-02181090

Submitted on 11 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient Recovery Path Computation for Fast Reroute in Large-scale Software Defined Networks

Kun Qiu, *Student Member, IEEE*, Jin Zhao, *Member, IEEE*, Xin Wang, *Member, IEEE*
Xiaoming Fu, *Senior Member, IEEE*, Stefano Secci, *Senior Member, IEEE*

***Abstract*—With an increasing demand for resilience in software-defined networks (SDN), it becomes critical to minimize service recovery delay upon route failures. Fast reroute (FRR) mechanisms are widely used in IP and MPLS networks by computing the recovery path before a failure occurs. The centralized control plane in SDN can potentially enhance path computation, so that FRR path computation can better scale in SDN than in traditional networks. However, traditional FRR path computation algorithms could lead to poor performance in large-scale SDN. The problem can become more severe for a highly dynamic network, which often sees dozens of failures or configuration changes in any single day. We propose a new algorithm that exploits pruned searching to quickly compute recovery paths for all-pair switches/hosts upon a link failure. For applications requiring stringent path robustness levels, we also extend this algorithm to quickly find the shortest guaranteed-cost path, which ensures that the recovery path used upon on-path link failures has the minimum cost. Compared with traditional solutions, our evaluations show that our algorithm is about $8 \sim 81$ times faster than the practical implementation, $1.93 \sim 3.11$ times faster than the state-of-the-art solution. Our results also show that the shortest guaranteed-cost path can reduce the cost of the recovery path significantly. Moreover, we design a prototype to show how to deploy our algorithm in an OpenFlow network.**

***Index Terms*—Software-Defined Networks, Path Computation, Fast Reroute**

I. Introduction

RECENTLY, *software-defined networks* (SDN) are increasingly deployed in enterprise, campus and carrier networks [1], [2]. By separating the control plane from the data plane, SDN has emerged as a promising paradigm with attractive features, such as switch programmability and centralized global view [3]. Although a large space of new functionalities in SDN has been explored, the problem of resilience to connectivity disruptions in SDN still remains challenging.

Measurement studies have revealed that link and node failures are quite frequent and unexpected in both data-center and wide-area network environments [4], [5]. Even if an important portion of failures are transient and last only few seconds or minutes as reported in [5], many delay-sensitive applications like VoIP or video conference [6] can tolerate only sub-50ms recovery times [7], [8]. Slow convergence during link/node failures can cause severe disruptions for these applications. Accordingly, fast reactions to such failures is a critical task.

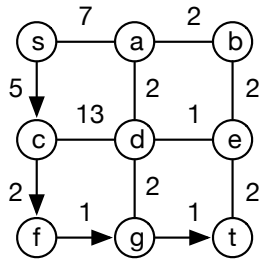
The procedure of computing and bypassing the traffic from the path with failures to a recovery path is referred to as rerouting [9]. The default SDN rerouting scheme includes the following steps: 1) failure report from switch to controller; 2) reactive computation of the recovery path at controller, and 3) flow table updates in the involved switches. The flow table update delay can be minimized remarkably [10]. However, path computation and switch-controller interaction still take a significant amount of the total recovery time. It is reported that the recovery time of *IGP* in SDN may be slower than in legacy network due to these overheads [11].

In general, the recovery paths can be reactively computed upon a failure event or proactively computed before a failure occurs. Reactive approaches might take significant time in finding an alternative path due to the route update delay. To allow rerouting as quick as possible, Fast Reroute (FRR) is proposed as a proactive recovery path pre-computation mechanism to initiate local switching tables. There have been considerable efforts using FRR towards minimizing rerouting time in traditional IP and MPLS networks [12]–[15]. FRR is also believed to be easily deployable in SDN [16]–[19]. There have been efforts in applying FRR to SDN using source routing [20]–[22]. However, as summarized in section II-A, traditional recovery path computation approaches suffer from prohibitive performance degradation in SDN environments as the scale of the network is much larger (typically more than $1,000$ switches/hosts [23], [24]) than traditional FRR scenarios. Meanwhile, it is reported that if failure recovery relies only upon built-in mechanisms of commercial OpenFlow switches, it may require up to hundreds of milliseconds [25]. As SDN is sensitive against single link failure, a full (100%) coverage protection solution is desired [26]. Therefore, it is vital to improve the efficiency of computing recovery paths for all-pair switches/hosts upon a link failure.

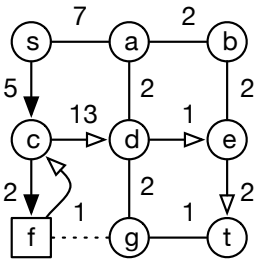
In this paper,we focus on the algorithmic challenges towards efficient FRR recovery path computation upon link failures in large-scale SDNs. With an FRR mechanism, the traffic detour locally occurs, with a recovery path from the **node with a failed link** (the last reachable node along the original path) to the destination node. There are different criteria for selecting the recovery path. A commonly used strategy is to choose the shortest path that has the minimum cost from

Manuscript received January 30, 2019; revised June 17, 2019; accepted June 27, 2019. The work was supported in part by Natural Science Foundation of China under Grant 61571136, by 863 program under Grant 2015AA016106, and by the European Unions Horizon 2020 MSCA COSAFE project (Grant No 824019).(Corresponding author: Jin Zhao.)

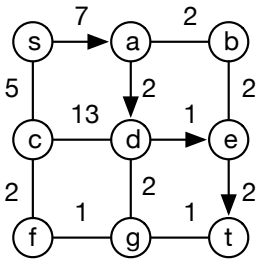
K. Qiu, J. Zhao, and X. Wang are with the School of Computer Science, Fudan University, Shanghai 201203, China, and also with Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 200433, China (e-mail:qkun@fudan.edu.cn; jzhao@fudan.edu.cn; xinw@fudan.edu.cn).
X. Fu is with Georg-August-Universitaẗ Goẗttingen, Germany (e-mail:fu@cs.uni-goettingen.de).
S. Secci is with Cnam, Cedric Lab, Paris, France (e-mail:seccis@cnam.fr).



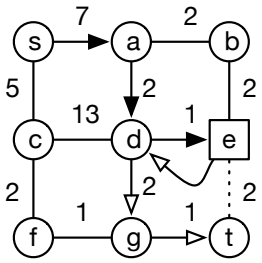
(a) The shortest path is $s \rightarrow c \rightarrow f \rightarrow g \rightarrow t$. The cost is 9.



(b) The guaranteed cost of the path in (a) is 25, which is incurred when link $f \rightarrow g$ failed in the worst recovery scenario. The last reachable node is f and the shortest path from f to t is $f \rightarrow c \rightarrow d \rightarrow e \rightarrow t$.



(c) A path from $s \rightarrow a \rightarrow d \rightarrow e \rightarrow t$. The cost is 12, a bit larger than the shortest path.



(d) The guaranteed cost of the path in (c) is 14, which is incurred when link $e \rightarrow t$ failed. The last reachable node is e and the shortest path from e to t is $e \rightarrow d \rightarrow g \rightarrow t$.

Fig. 1. A shortest path may lead to a high recovery cost if one of its edges failed. Thus, finding the path that has the shortest guaranteed cost can reduce the cost of the path in the worst recovery scenario. In the example, the guaranteed cost of the shortest path cannot meet the cost constraint if the notified constraint is smaller than 14.

the failed node to the destination node [27], [28], which is called the *shortest recovery path*. This work aims to provide a fast path computation approach for: 1) shortest recovery path, and 2) shortest guaranteed-cost path. For the first challenge, we need to efficiently compute shortest recovery paths in a graph. The shortest recovery path can be computed by utilizing the shortest path algorithms between the failed node and the destination. Dijkstra is the default algorithm used in current carrier-grade controllers such as ONOS [29] and OpenDaylight [30]. Since the naïve Dijkstra is not scalable for large-scale networks, various approaches [31], [32] have been proposed to tackle the scalability of shortest recovery path computation. The most efficient algorithm [28] can achieve a time complexity of $O(M \log N + N^2)$ for single-pair shortest recovery path computation in an undirected graph with N nodes and M edges.

For the second challenge, we need a fast algorithm for computing the shortest guaranteed-cost path. The *guaranteed cost* is defined as the cost of the worst-case recovery path associated with a given path (i.e., the maximum recovery cost upon an on-path link failure). A path P that has the minimum guaranteed cost is called the shortest guaranteed-cost path. We use Fig. 1 to illustrate the necessity of finding the shortest guaranteed-cost path. Intuitively, the shortest guaranteed-cost path is more preferable in some circumstances because recovering the link failure on the shortest path may lead to high cost. An algorithm is proposed [28] with complexity $O(MN+N^2 \log N)$ to compute the all-pair shortest guaranteed-cost path. However, its performance is not efficient enough for large-scale networks.

We propose an efficient two-stage algorithm, which includes indexing stage and querying stage, to address the first challenge by utilizing the general *pruned searching* [33], [34] framework with our own *pruning strategy* to index intermediate results. Our algorithm can achieve significant performance improvement by reducing the indexing time complexity of existing solutions to $O(lM + lN(logN + l))$ with querying time complexity $O(l)$, where l, a factor introduced by pruned searching, is proved as a pretty stable integer less than 300 in most real-world topologies. Intuitively speaking, a lower l indicates a more efficient *pruning strategy* in the search, which means that a higher number of unnecessary intermediate results are pruned during indexing. We compare our solution with both theoretical state-of-the-art and the practical implementation (Dijkstra algorithm, commonly used in network system design) in all-pairs nodes (100% nodes protection) scenarios. The computation time of our solution is $8 \sim 81$ times faster than the practical solution in a variety of networks with the scale range from $153 \sim 1,163$ nodes (from Topology Zoo [35] and AS network [36]) , and it is $1.93 \sim 3.11$ times faster than a theoretical state-of-the-art (optimized pre-computing) solution with $754 \sim 5,340$ nodes in all-pairs nodes protection scenario. From our evaluation results, the practical solution cannot finish evaluations with more than $1,163$ nodes networks, and the optimized pre-computing solution may finish evaluations with more than $5,340$ nodes networks in several days. On the contrary, our algorithm can compute all-pairs recovery path range from several minutes to several hours. Also, our algorithm can provide much higher efficiency in the scenario that the percentage of nodes protection is less than 100%. Moreover, we design a prototype to show how to deploy our algorithm in an OpenFlow network. The evaluation result indicates that Fast Reroute is far faster than the conventional approaches in OpenFlow network. Meanwhile, we extend the first algorithm to address the shortest guaranteed-cost path with the same time complexity. We prove the correctness of our pruning strategy and evaluate it with real-world topologies [35], [36]. We also compare the guaranteed cost between the shortest path and the shortest guaranteed-cost path. Our results show the shortest guaranteed-cost path can significantly reduce the recovery cost in the worst case upon a link failure in real-world topologies. Furthermore, we also give some discussion on multiple link failures and nodes failures.

The rest of paper is organized as follows. We first summarize the background and motivation in section II, and then present the problem statement in section III. In section IV,

we introduce our algorithm for computing shortest recovery path, and we introduce our algorithm for computing shortest guaranteed-cost path in section V. In section VI, we evaluate our algorithms and analyze the evaluation results, and we discuss the case of multiple links/node failure, and some implementation and experimentation issues. Finally, we conclude the paper in section VII.

II. Background and Motivation

After resuming the state of the art on fast rerouting and path recovery in IP networks, we provide the rationale and motivation of our proposal.

A. Related Work

*1) **FRR in IP and MPLS network**:* Applying FRR in IP and Multi-Protocol Label Switched (MPLS) networks [13], [14] can reduce the recovering time remarkably when a failure occurs. The loop-free alternate (LFA) [15] is the simplest method to provide FRR in IP networks and it is widely used in commodity devices. MPLS FRR [37] uses RSVP-TE [38] to maintain recovery path. However, since SDN is based on a centralized controller, these solutions do not work out-of-the-box in SDN [39]. This because these solutions are designed for pure distributed network environment.

*2) **Existing FRR solutions in SDN**:* Existing works utilize the shortest path algorithm [27], [40], [41] to compute the FRR recovery path. Most of SDN controllers including ONOS [29] integrate Dijkstra algorithm as the primary shortest path computation algorithm. However, with the increase in network scale [42], Dijkstra is not as efficient as it performs in small-scale networks. McNettle [23], a multicore supported controller is designed to scale up to large networks with $1,000$ switches; and ONOS, the carrier-grade controller, can support even more switches than existing single server designs by utilizing distributed computing. But as evidence, the running time is more than several seconds for an $O(N^3)$ algorithm (e.g., computing recovery path for all-pairs nodes by Dijkstra) when N is larger than $5,000$ with an Intel i7-920 @2.66GHz processor [43]. Moreover, it is also reported that in ONOS, the path computation cannot achieve the expected throughput [29] since ONOS does not utilize distributed computing in path computation [44]. There are also previous attempts to address FRR in the OpenFlow network. A heap-Dijkstra based system [45] utilizes the Group Table feature to deploy fast failover entries into the OpenFlow switch. Moreover, a hybrid recovery path computation algorithm [46] has been proposed to compute the recovery path both on node and link failure. However, both schemes show no advantages in path computation complexity with respect to the traditional Dijkstra algorithm..

*3) **Replacement path problem/Parallel shortest path algorithm**:* Computing shortest recovery problem is a subproblem of the well-studied *replacement path* problem in graph theory. In solving *replacement path* problem, an $O(M + N\log N)$ algorithm is proposed by [31] for a given pair of s and t in an undirected graph. It has been proven that there is no solution faster than $O(MN + N^2 \log\log N)$ [32] in directed graphs. Clearly, directly applying these algorithms to all-pair recovery path computation in SDN is unacceptable due to the high time complexity. Though there exist parallel shortest path algorithms, e.g. $\Delta-$stepping algorithm [47] with linear time complexity in directed graphs, most existing SDN controllers can not support distributed path computation at multiple controller instances. For example, ONOS can support multiple controller instances, but the path computation is still performed at one instance.

B. Design rationale

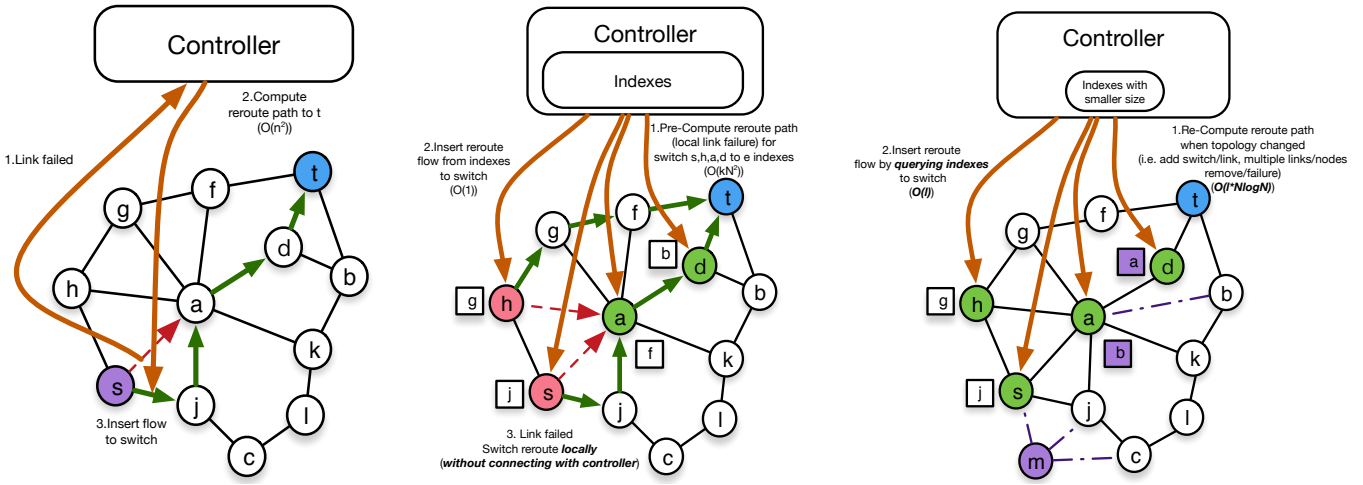
Our FRR mechanism seeks to reduce recovery path computation by separating the computation into two stages: indexing stage and querying stage. Different from the conventional approach that directly computes final best paths (all-pair shortest recovery path or shortest guaranteed-cost path), we only compute intermediate results with an optimized algorithm in the indexing stage, and then compute the final paths by using these indexed intermediate results in the querying stage. Moreover, we propose a strategy in our optimized algorithm that can significantly reduce unnecessary intermediate results in the indexing stage. In this way, we can sharply decrease the computation time, which is the sum of the indexing time (time used in indexing stage) and querying time (time used in querying stage) for all-pair shortest recover paths.

In order to motivate our design, we use Fig. 2 to show a case for path recovery under link failure. In this example, different recovery schemes are deployed to protect the path from s to t. Fig. 2(a) shows the traditional reactive scheme for failure recovery in SDN. When a link fails, the switch firstly sends a recovery request to the controller, then the controller computes a new path and inserts forwarding rules into the switch. Our experiments reveal that for a $1,163$-node network, the computing time needs 195ms for one single shortest recovery path by utilizing the Dijkstra algorithm.

Fig. 2(b) shows the traditional Fast Reroute algorithm in SDN [27]. The controller computes the recovery path *before failure occurs*, and inserts forwarding rules into switches (s,h,a,d) that need be protected. When the failure occurs, such as $h \rightarrow a$ and $i \rightarrow a$ failed, the reroute $h \rightarrow g$ and $i \rightarrow j$ occurs in the switch locally. For a $1,163$-node network, protecting paths for all switches to t (when $k = N$ in the time complexity) needs 227.231s.

We also leverage a proactive approach to precomputing the recovery path *before failure occurs*. However, our method does not compute the recovery path directly. Instead, as shown in Fig. 2(c), the controller computes the intermediate results as indexes in the indexing stage, and query recovery paths based on the index for protected switches in the querying stage. When a failure occurs, the protected switch can redirect the flow to the recovery path without controller intervention. If the topology changed, i.e., multiple switch/link add/remove/failures, our method can re-compute indexes and recovery paths rapidly. For a $1,163$-node network, indexing only needs 0.144s, computing recovery paths for all switches to t needs only 2.785s.

The key intuition exploited by our design is that we can accelerate the recovery path setup by precomputing the inter-



(a) Reactive failure recovery in SDN.

(b) Traditional Fast Reroute algorithm in SDN [27]. All recovery path are stored in indexes.

(c) Our efficient Fast Reroute algorithm in SDN. Only necessary intermediate results are stored in indexes

Fig. 2. The differences between traditional failure recovery and our method in SDN. We suppose the path from s, h, a, d to t are protected. We use a square with a node near the switch to indicate the next hop of backup path to t, which is the path when the first link of the default path failed.

mediate indexing results. To this end, we profitably separate the computation into two stages: indexing stage and querying stage. Another concern in the design is the space overhead in storing the index. One naïve approach is to compute all the final recovery paths and store them in the index. Query can then be achieved with an $O(1)$ time (directly access the computed results in memory after the computing is finished). However, the index size is unacceptable for large-scale networks. Motivated by *2-hop cover* strategy [48], we design a pruned searching algorithm to reduce the index size by balancing the *indexing* time and *querying* time. Our method can significantly reduce the size of indexes by pruning unnecessary results during indexing. The tradeoff is to slightly increase the computation time when querying the recovery paths. Intuitively speaking, our method computes the index iteratively and prune the intermediate results that are never used after each iteration. Final path results can then be computed on the pruned index. The example of Fig. 3, for instance, shows the differences in index size between the naïve method and our method.

In the following sections, we will describe the problem formally.

III. Problem Statement

In the following we present the mathematical notations used in the paper and give a formal definition of the addressed problem.

A. Notations

We describe our notations which are used in this paper in TABLE I. In this paper, we describe a network as an undirected graph $G = (V, E)$, with node set V and edge set E. N indicates the number of nodes and M indicates the number of edges. We denote an edge from node u to node v by $e_{u,v}$. Each edge is associated with a cost (e.g., delay or distance) between two nodes. We use $\sigma(e_{u,v})$ to indicate the cost of $e_{u,v}$.

Let $P(s,t)$ be a path from node s to node t. The internal nodes are the nodes in $P(s,t)$ excluding s and t. We also use $P'^{(s,t)}(u,v)$ to denote a subpath in $P(s,t)$. Let $\sigma(P(s,t))$ indicate the cost of $P(s,t)$ (e.g., delay of the distance of a path) between node s and t.

We can connect two paths into one if the destination of the first path is the same as the source of the second path, and we can easily obtain the cost of the connected path by adding up the cost of the two paths.

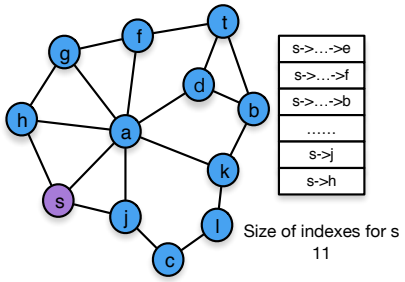
B. Problem definition

As we mentioned above, the paper studies the following two problems: **1) Computing the shortest recovery path $R(P(s,t))$ for a given node pair(s,t)** and **2) Computing the shortest guaranteed-cost path $Gp(s,t)$ for a given node pair (s,t).**

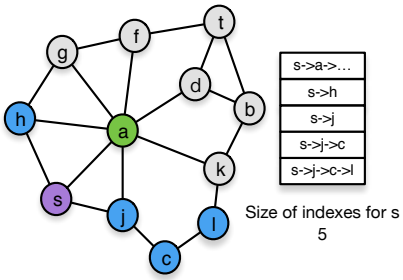
Briefly speaking, the recovery path $R(P(s,t))$ of a specific path $P(s,t)$ is another path from s to t, which does not pass through the first edge in $P(s,t)$. This path is used for rerouting network packets from s to t when the first link of the original path fails. As we have mentioned above, with an FRR mechanism, the rerouting occurs locally on the switch. In other words, when a link failure occurs on a switch, the switch chooses the recovery path to the destination locally to bypass the failed link that is the first link of the original path from the switch to the destination.

Problem 1 seeks to compute the shortest recovery path for any path between an arbitrary pair of nodes in graph G.

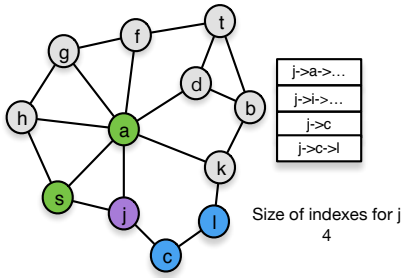
For problem 2, we first define the guaranteed cost $\eta(P(s,t))$ as the cost of the actual path $P(s,t)$ followed by a packet in the worst-case recovery scenario. Because $P(s,t)$ can cause different recovery costs upon different link failure scenarios, there will be one failure scenario that leads to the maximum



(a) If we directly compute recovery paths for one node, each node needs 11 indexes (from source node to all other 11 nodes).



(b) If we already indexed some intermediate results, such as recovery paths from a to other nodes, we just connect $s \rightarrow a$ with recovery paths start from a to get recovery paths that need pass through a from s. We only need index recovery paths that do not need pass through a. Only 5 indexes are needed for s.



(c) If recovery paths of s and a are indexed, we only need index recovery paths that do not need pass through s and i, Only 4 indexes are needed for j.

Fig. 3. The differences in index size between the naïve method (a) and our method (b) (c).

TABLE I
TERMS OF DEFINITION

Notation	Description
$G = (V, E)$	A graph G with node set V and edge set E
$e_{u,v} \in E$, $u, v \in V$	An edge from u to v
$\sigma(e_{u,v})$	The cost of an edge $e_{u,v}$
$P(s,t)$	A path from s to t
$P'^{(s,t)}(u,v)$	$P'^{(s,t)}(u,v)$ is a subpath of $P(s,t)$
$\sigma(P(s,t))$	The cost of a path $P(s,t)$
$R(P(s,t))$	The shortest recovery path for $P(s,t)$
$\eta(P(s,t))$	The guaranteed cost of $P(s,t)$
$Gp(s,t)$	The shortest guaranteed-cost path from s to t

TABLE II
TERMS OF CONSTRAINED PATH

Notation	Description
$P^{>u}(u,v)$	A path set from u to v, whose internal nodes are larger than u
$P_{*,*}(u,v)$	A path set from u to v, without any constrained edges ('*' means ignoring the constraint of edge)
$P_{=x,=y}(u,v)$	A path set from u to v, whose first edge must be $e_{u,x}$ and last edge must be $e_{y,v}$
$P_{\neq x,*}(u,v)$	A path set from u to v, whose first edge cannot be $e_{u,x}$ (ignoring the constraint of last edge)
$P_{*,\neq y}(u,v)$	A path set from u to v, whose last edge cannot be $e_{y,v}$ (ignoring the constraint of first edge)
$P_{\nexists e_{x,y}}(u,v)$	A path set from u to v, whose edges cannot exist $e_{x,y}$

recovery cost, say $\eta(P(s,t))$. As we have mentioned above, the shortest guaranteed-cost path $Gp(s,t)$ is a path from s to t, which has the smallest guaranteed cost.

We formally define two problems as follows:

1) **Computing the shortest recovery path $R(P(s,t))$ for given (s,t)**: given a graph $G(V,E)$ and a path $P(s,t)$ between an arbitrary pair of nodes, return the shortest path from s to t in $G(V, E - e_{s,n_1})$, where e_{s,n_1} is the first edge in $P(s,t)$.
2) **Computing the shortest guaranteed-cost path $Gp(s,t)$ for given (s,t)**: given a graph $G(V,E)$ and an arbitrary pair of nodes s, t, return the path from s to t that satisfies the following statement: $Gp(s,t)$ has the smallest guaranteed cost.

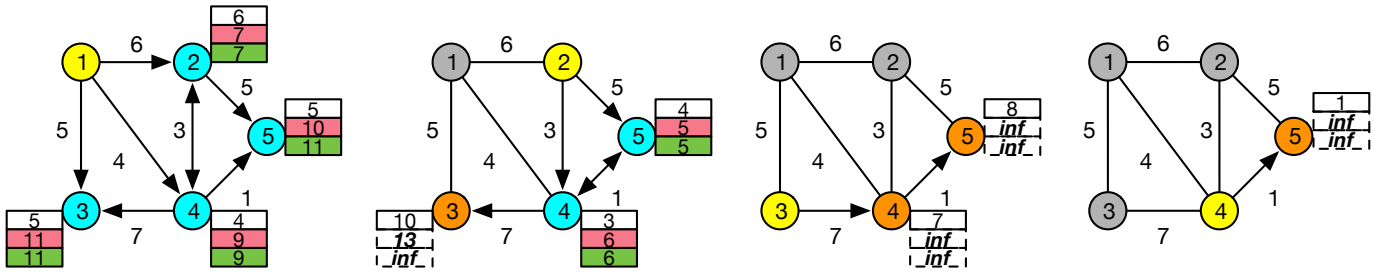
IV. Shortest Recovery Path

As we have mentioned above, the efficiency of our algorithm is brought by reducing the amount of computation that is proved unnecessary. We separate our computation into two stages: indexing stage and querying stage; we prove that the most unnecessary computation tasks can be avoided thanks to the indexing stage.

In this section, we will first describe the algorithm in the querying stage: **How to query the shortest recovery path with the knowledge of indexed intermediate results.** Then, we will describe the algorithm in the indexing stage, which is the core contribution of our proposing: **How to index intermediate results, and reduce unnecessary results (we call pruned searching) with pruning strategy.**

A. Query shortest recovery path with 2-hop method

Firstly, we introduce how to utilize *2-hop cover* [48] in querying shortest recovery path. Intuitively speaking, the *2-hop cover* provides the ability to compute a complicated path (with some edge/node constraints) by combining the results of two simple path computation (such as the shortest path). We give the notation of the constrained path in TABLE II. Suppose that each node in V is assigned with a node ID in an integer. Thus, we can compare two nodes u, v in V by $u > v$ or $u \leq v$. Let $P(s,t) = \{s, n_1, n_2, \ldots, n_m, t\}$ and the



(a) The first BFS starts from node 1 to compute $P^{>1}(1,v)$. We search all nodes. The node 5 has cost 11 in green square since the recovery path $1 \to 2 \to 4 \to 5$ have the same last edge with the shortest path $1 \to 4 \to 5$. We have to use path $1 \to 2 \to 5$ as the minimum cost of the path that does not pass through the last edge of shortest path.

(b) The second BFS starts from node 2 to compute $P^{>2}(2,v)$. Since path $2 \to 5 \to 4 \to 3$ is longer than $2 \to 1 \to 3$ that $1 \to 3$ was indexed in the first BFS, we prune this path whose cost (13) should be in red square originally. Also, we use *'inf'* in green square since we cannot find a path to node 3 that does not pass through $4 \to 3$.

(c) The third BFS starts from node 3 to compute $P^{>3}(3,v)$. The cost in red and green square of node 4 and 5 are pruned since we do not find a path without passing through the first edge or the last edge of the shortest path.

(d) The last BFS starts from node 4 to compute $P^{>4}(4,v)$. Only the shortest path $4 \to 5$ is indexed.

Fig. 4. An example of our algorithm in the indexing stage. The graph is an undirected graph, and the arrow indicates the search direction in the BFS. Yellow nodes denote the root in each BFS, blue nodes denote the nodes we have searched, orange nodes indicate the nodes we have searched but some results (paths) are pruned, grey nodes denote the nodes which are already used as roots, to ensure blue nodes are larger than the root in current BFS. The number in the white square indicates the cost of the shortest path from the yellow node. The number in the red (green) square indicates the minimum cost of the path that does not pass through the first (last) edge of the shortest path. After all BFS finished, we can query all-pair shortest recovery path in the querying stage. A query example: $\text{QUERY}(2,5,e_{2,4})$=$min\{P^{>2}_{\neq 4,*}(2,5), P^{>1}_{*,\neq 4}(1,2) + P^{>1}_{\nexists e_{2,4}}(1,5)\}$=$min\{5, 6+5\}$ = 5.

first edge of $P(s,t)$ is e_{s,n_1}. Without loss of generality, we assume $s < t$. We can swap s with t if $s > t$. We denote the cost of a path between node s and t by $\sigma(P(s,t))$.

We can separate the recovery path of $P(s,t)$ into two parts. We define $P^{>u}_{=x,=y}(u,v) = \{u, x, \ldots, y, v\}$ as the path set that belongs to $P^{>u}(u,v)$ with two constraints: the path must start with $e_{u,x}$ and end with $e_{y,v}$. If we change '$=$' to '$\neq$', the meaning of the constraint turns to be that the path cannot start with $e_{u,x}$ or end with $e_{y,v}$. We also use the wildcard '$*$' to indicate that the constraint can be ignored. Thus, $P^{>u}_{*,*}(u,v) = P^{>u}(u,v)$. We use $P^{>u}_{\nexists e_{x,y}}(u,v)$ to represent the path in $P^{>u}(u,v)$ that does not pass through $e_{x,y}$.

Suppose we have computed all $P^{>u}(u,v), \forall u,v \in V$ in the indexing stage. With the knowledge of all $P^{>u}(u,v), \forall u,v \in V$ available, we can compute the cost of the shortest recovery path by defining the QUERY function: $\forall n_q \in V$ and $e_{s,n_1} \in P(s,t)$, note that e_{s,n_1} is the first edge in $P(s,t)$.

$$\sigma(R(P(s,t))) = \text{QUERY}(s,t,e_{s,n_1}) =$$
$$min \begin{cases} \sigma(P^{>n_q}_{*,\neq n_1}(n_q,s)) + \sigma(P^{>n_q}_{\nexists e_{s,n_1}}(n_q,t)), & (\exists n_q < s, n_q < t) \\ \sigma(P^{>n_q}_{*,\neq n_1}(n_q,s)) + \sigma(P_{\nexists e_{s,n_1}}(n_q,t)), & (\exists n_q \geq t) \\ \sigma(P^{>s}_{\neq n_1,*}(s,t)). \end{cases}$$

The output of the $\text{QUERY}(s,t,e_{s,n_1})$ is the cost of shortest recovery path of $P(s,t)$. The sum of cost in the first equation indicates the minimum cost of the sum of two paths $P(n_q,s) \in P^{>n_q}_{*,\neq n_1}(n_q,s)$ with $P(n_q,t) \in P^{>n_q}_{\nexists e_{s,n_1}}(n_q,t)$ by finding $n_q, \exists n_q < s, n_q < t$. The second equation is the same as the first one. If we cannot find a common n_q for $P^{>n_q}_{*,\neq n_1}(n_q,s)$ and $P^{>n_q}_{\nexists e_{s,n_1}}(n_q,t), (\exists n_q < s, n_q < t)$, the sum of cost is ∞; similarly, if we cannot find a common n_q for $P^{>n_q}_{*,\neq n_1}(n_q,s)$ and $P_{\nexists e_{s,n_1}}(n_q,t), (\exists n_q \geq t)$, the sum of cost is ∞; if we cannot find a path in $P^{>s}_{\neq n_1,*}(s,t)$, the cost of $P^{>s}_{\neq n_1,*}(s,t)$ is ∞ too.

In order to estimate the time complexity, we define $L(v) = \{P(u,v) | P(u,v) \in P^{>u}(u,v), \forall u \in V\}$. $L(v)$ is the path set for all paths whose destination is v. For each node v, if we store the $L(v)$ in an order sorted by u, we can compute the $\text{QUERY}(s,t,e_{s,n_1})$ in $O(|L(s)| + |L(t)|)$ by a merge-join-like algorithm [33].

B. Index intermediate results with pruning strategy

As we have mentioned, we can use QUERY to compute the shortest recovery path in computed $P^{>u}(u,v), \forall u,v \in V$. For computing $P^{>u}(u,v)$ with a given u, we can use a standard Breadth-First-Search(BFS) [49] algorithm, which starts from the head of a queue and expends all of its adjacent nodes into the queue. The queue initialized with root u, and the BFS will stop if no new path found. Moreover, we add an expanding constraint in BFS algorithm to ensure each expanded node is larger than u. However, directly computing $P^{>u}(u,v)$ is a brute force method, which leads to a significant amount of space and time. Motivated by utilizing pruned searching in efficiently computing shortest path and *k-th* shortest path [33], [34], we believe that most paths in $P^{>u}(u,v)$ are unnecessary for computing shortest recovery path.

Thus, utilizing a pruning strategy in computing $P^{>u}(u,v)$ to prune unnecessary results and reduce the indexing time, is necessary. This framework is called *pruned searching*.

Proposition 1. $\forall u,v \in V$, and we denote $e_{u,n_a}, e_{n_z,v} \in P(u,v)$. Only the following paths

$$\begin{cases} \{P^{\alpha}(u,v) | \sigma(P^{\alpha}(u,v)) = min\{\sigma(P^{>u}_{*,*}(u,v))\}\}, \\ \{P^{\beta}(u,v) | \sigma(P^{\beta}(u,v)) = min\{\sigma(P^{>u}_{*,\neq n_z}(u,v))\}\}, \\ \{P^{\gamma}(u,v) | \sigma(P^{\gamma}(u,v)) = min\{\sigma(P^{>u}_{\neq n_a,*}(u,v))\}\}. \end{cases}$$

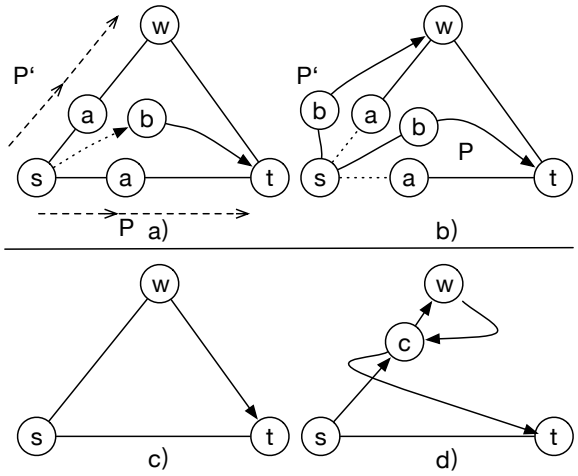


Fig. 5. Different cases in pruning strategy. In Case a), suppose the original path is from $s \to b \to t$, and the shortest path from s to t is $P' : s \to a \to w \to t, (\exists w < s, w < t)$ or $P : s \to a \to t$. If $s \to b$ failed, the shortest recovery path must be one of P' or P. In Case b), the original path is from $s \to b \to t$ or $s \to b \to w \to t, (\exists w < s, w < t)$, while $s \to b$ is the first edge of the shortest path from s to t. Thus, if $s \to b$ failed, the shortest recovery path must be the shortest path from s to t without passing through $s \to b$. There are two types of shortest recovery paths: $P : s \to a \to t$ or $P' : s \to a \to w \to t, (\exists w < s, w < t)$. Case c) and Case d) can ensure there is no other path needed if $w \to t$ intersects in $s \to w$. In Case d), if $w \to t$ intersects in $s \to w$, we can find a new recovery path shorter than $s \to w \to t$.

are needed for computing $R(P(s,t))$, $\forall s,t \in V$ in the querying stage,

with s,t limitations:

$$\forall s,t \in V, \exists n_q < s, n_q < t,$$
$$\sigma(P^{>n_q}_{*,\neq n_1}(n_q,s)) + \sigma(P^{>n_q}_{\nexists e_{s,n_1}}(n_q,t)) \neq \infty$$

We use Fig. 5 to intuitively explain how Proposition 1 works. w is a n_q that satisfies the limitation. As we have defined, the shortest recovery path ($s \to a \to t$ or $s \to a \to w \to t$) of the original path ($s \to b \to t$ or $s \to b \to w \to t$) from s to t must be the shortest path from s to t without passing through the first edge of the original path. **Case a):** If the first edge of the original path from s to t is different from the first edge of the shortest path from s to t, we can choose the shortest path (can be computed with $P^\alpha(s,t)$) as the shortest recovery path. **Case b):** Otherwise, if the first edge of the original path from s to t is the first edge of the shortest path from s to t, we need to choose the path whose first edge is different from the shortest path from s to t (can be computed with $P^\gamma(s,t)$, $P^\beta(t,s)$ if $s > t$) as shortest recovery path.

Case c) and Case d) can ensure there is no other path needed if $w \to t$ intersects in $s \to w$. Respectively. In **Case c)**, if $w \to t$ is disjoint with $s \to w$, $w \to t$ must be the shortest path for ensuring the recovery path $s \to w \to t$ to be the shortest. In **Case d)**, if $w \to t$ intersects in $s \to w$, we assume the joint point is c and we can find the shortest recovery path $s \to c \to w \to c \to t$ is longer than $s \to c \to t$. In other words, if $w \to t$ intersects in $s \to w$, we can find a new recovery path shorter than $s \to w \to t$.

Formally, we provide the proof of correctness as follow:

Proof 1. Suppose we have computed $P^\alpha(u,v)$, $P^\beta(u,v)$, $P^\gamma(u,v)$ with $\forall u,v \in V$. We denote the first edge of $P^\alpha(u,v)$, $P^\gamma(u,v)$ is e_{u,n_α}, e_{u,n_γ} and the last edge of $P^\beta(u,v)$ is $e_{n_\beta,v}$.

Let the original path be

$$P(s,t) = \{s, n_1, n_2, \ldots, n_m, t\}.$$

If we suppose $e_{s,n_1} \in P(s,t)$, when $e_{s,n_1} \neq e_{s,n_\alpha}$, $\forall n_q \in V$ we can get

$$\sigma(R(P(s,t))) = \textsc{Query}(s,t,e_{s,n_1}) = \sigma(P^\alpha(s,t)), \quad (1)$$

otherwise, when $e_{s,n_1} = e_{s,n_\alpha}$, $\forall n_q \in V$, $\exists n_q < s, n_q < t$, we can get

$$\sigma(R(P(s,t))) = \textsc{Query}(s,t,e_{s,n_1}) =$$
$$min \begin{cases} \sigma(P^{>n_q}_{*,\neq n_1}(n_q,s)) + \sigma(P^{>n_q}_{\nexists e_{s,n_1}}(n_q,t)) \quad (\exists n_q < s, n_q < t), \\ \sigma(P^\gamma(s,t)). \end{cases} \quad (2)$$

Suppose $\exists n_q \in V, n_q < s, n_q < t$ and

$$\sigma(P(s,t)) = \sigma(P^{>n_q}_{*,=n_\alpha}(n_q,s)) + min\{\sigma(P^{>n_q}(n_q,t))\},$$

then we can get

$$min\{\sigma(P^{>n_q}_{*,\neq n_1}(n_q,s))\} = \sigma(P^\beta(n_q,s)). \quad (3)$$

to make the first equation of QUERY minimum.

We denote $P^\delta(n_q,t) = \{P^\delta(n_q,t) | \sigma(P^\delta(n_q,t)) = min\{\sigma(P^{>n_q}_{\nexists e_{s,n_1}}(n_q,t))\}\}$. Suppose $\exists n_q \in V$, $n_q < s, n_q < t$. $P^\alpha(n_q,t)$ is disjoint with $P^\beta(n_q,s)$, we can get

$$P^\delta(n_q,t) = P^\alpha(n_q,t). \quad (4)$$

otherwise $\exists n_q \in V, n_q < s, n_q < t$, $P^\alpha(n_q,t)$ must intersect in $P^\beta(n_q,t)$, we denote the joint point as n_j, and $\forall n_p >= n_q$. From Equation (2) (same as Equation (1)) we can get,

$$\begin{aligned}
&min\{\sigma(P^{>n_q}_{*,\neq n_1}(n_q,s)) + \sigma(P^\delta(n_q,t))\} \\
&= min\{\sigma(P^{>n_q}_{*,\neq n_1}(n_j,s)) + \sigma(P^{>n_q}(n_j,n_q)) \\
&\quad + \sigma(P^{>n_q}(n_q,n_j)) + \sigma(P^{>n_q}(n_j,t))\} \\
&\geq min\{\sigma(P^{>n_q}_{*,\neq n_1}(n_j,s)) + \sigma(P^{>n_q}(n_j,t))\} \\
&\geq min\{\sigma(P^{>n_p}_{*,\neq n_1}(n_p,s)) + \sigma(P^{>n_p}(n_p,t))\} \\
&= min\{\sigma(P^{>n_p}_{*,\neq n_1}(n_p,s)) + \sigma(P^\alpha(n_p,t))\} \\
&= min \begin{cases} \sigma(P^\alpha(n_p,s)) + \sigma(P^\alpha(n_p,t)), (e_{n_a,s} \notin P^\alpha(n_p,s)) \\ \sigma(P^\beta(n_p,s)) + \sigma(P^\alpha(n_p,t)). (e_{n_a,s} \in P^\alpha(n_p,s)) \end{cases}
\end{aligned} \quad (5)$$

As a conclusion from Equation (1),(2),(3),(4) and (5), Proposition 1 is proven. □

However, $\exists n_q \geq t$, $\sigma(P_{\nexists e_{s,n_1}}(n_q,t))$ (in the second equation of QUERY) may not be computed by $P^\alpha(u,v)$, $P^\beta(u,v)$, $P^\gamma(u,v)$. Briefly speaking, suppose e_{s,n_1} exists in $P^\alpha(u,v)$, $P^\beta(u,v)$, $P^\gamma(u,v)$ simultaneously, only indexing $P^\alpha(u,v)$, $P^\beta(u,v)$, $P^\gamma(u,v)$ will result $\sigma(P_{\nexists e_{s,n_1}}(n_q,t)) = \emptyset$. Thus, we need to use Dijkstra algorithm to compute $P_{\nexists e_{s,n_1}}(n_q,t)$ and add it into indexes.

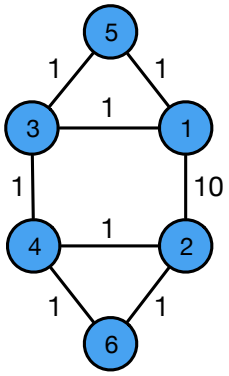


Fig. 6. $e_{3,4}$ exists in $P^{\alpha}(1,2)$, $P^{\beta}(1,2)$, $P^{\gamma}(1,2)$ simultaneously. We need Dijkstra algorithm to compute $P_{\sharp e_{3,4}}(1,2)$, and add it into indexes.

As an example in Fig. 6, when we QUERY$(3,2,e_{3,4})$, which means finding the recovery path from node 3 to node 2 without passing through $e_{3,4}$, we cannot find any path that satisfies the limitation of n_q. Suppose $n_q < t$, such as $n_q = 1$, then $P^{\alpha}(1,3) = \{1,3\}, P^{\alpha}(1,2) = \{1,3,4,2\}, P^{\beta}(1,2) = \{1,5,3,4,2\}, P^{\gamma}(1,2) = \{1,3,4,6,2\}$. $e_{3,4}$ exists in $P^{\alpha}(1,2)$, $P^{\beta}(1,2)$, $P^{\gamma}(1,2)$ simultaneously. $P^{>1}_{\sharp e_{3,4}}(1,2) = \emptyset$. Suppose $n_q > t$, if $n_q = 3$, $\sigma(P_{\sharp e_{3,4}}(1,2))$ cannot be computed via existing indexes. Thus, we need to use Dijkstra algorithm to compute $P_{\sharp e_{3,4}}(1,2)$, and add it into indexes.

Fortunately, this case seldom occurred in the real-world dataset from our evaluation result. In small-scale backbone network, the probability of this case is about 3% to 4%. In large-scale IPv4 and IPv6 network, the probability of this case is lower than 0.05%. Thus, introducing Dijkstra algorithm will not cause a significant decrease in efficiency.

The case A) can be explained by Equation (1) and case B) can be explained by Equation (2) and (3). We have also proved that there is no loop created in case B) by Equation (4) (5).

Proposition 1 provides the ability to prune paths whose cost are larger than $P^{\alpha}(u,v)$, $P^{\beta}(u,v)$ and $P^{\gamma}(u,v)$ in computing $P^{>u}(u,v)$ with a given u. They are initialized with ∞ at the beginning of the BFS, and they will be replaced by newer paths whose cost are smaller during the computation. We give an example of our algorithm in the indexing stage in Fig. 4. After the indexing stage finished, we can compute all-pair shortest recovery path by utilizing QUERY method for all-pair nodes.

C. Pruned searching algorithm

With Proposition 1, now we propose a search algorithm with the pruning strategy to only compute the necessary results in $P^{>u}(u,v)$. We utilize Algorithm 1—INDEX, a pruned searching algorithm to compute the $P^{>u}(u,v)$ with a given u. At last, we increase u gradually to compute all necessary $P^{>u}(u,v)$ by Algorithm 2.

It is easy to utilize the pruning strategy (Proposition 1) by implementing a method PRUNE. PRUNE(u,h) returns the result whether the new path $P(u,h)$ is smaller than current $P^{\alpha}(u,h)$, $P^{\beta}(u,h)$ and $P^{\gamma}(u,h)$. If it is larger than all of them, the new path can be pruned. Moreover, for further reducing the size of $P^{>u}(u,v)$, we can also prune the $P^{\alpha}(u,h)$, $P^{\beta}(u,h)$ or $P^{\gamma}(u,h)$ if the cost of $P^{\alpha}(u,h)$, $P^{\beta}(u,h)$ or $P^{\gamma}(u,h)$ is larger than the path with the path constraints (same as $P^{\alpha}(u,h)$, $P^{\beta}(u,h)$ or $P^{\gamma}(u,h)$), which can be computed in $P^{>u'}(s,t)$ $(\forall u' < u)$ by QUERY function. Thus, we also need the computed set $P^{>u'}(u',v)$ $(\forall u' < u)$ to filter the unnecessary path. We can utilize an algorithm that is similar to QUERY for obtaining the computed path in $P^{>u'}(u,h)$ $(\forall u' < u)$. Therefore, the time complexity of method PRUNE is $O(|L(u)| + |L(h)|)$ for one query by a merge-join-like algorithm. As mentioned earlier, $L(v)$ is the path set for all paths whose destination is v. After utilizing the pruning strategy, the average size of $L(v)$ has a significant decrease. We also use L to indicate the union of $L(v)$ $(\forall v \in V)$, which is the set of all computed $P^{>u}(u,v)$.

Algorithm 1: INDEX: Pruned searching for a given u to compute $P^{>u}(u,v)$

Input: Node u, existing path set $P^{>u'}(u',v)$ $(\forall u' < u)$
Output: The path set $P^{>u}(u,v)$
h is the expanded node in search
$h.cost$ is the cost from u to h
$Q \leftarrow$ a priority queue with one element u
$T \leftarrow \emptyset$
while *Q is not empty* **do**
  Dequeue h from Q
  if PRUNE(u,h) *is false* **then**
    A path $P(u,h)$ is computed
    $T \leftarrow T \cup P(u,h)$
    for *w that is the neighbor of h and $w > u$* **do**
      $w.cost \leftarrow h.cost + \sigma(e_{h,w})$
      Insert w to Q
Return T

Algorithm 2: Compute all $P^{>u}(u,v)$ by pruned searching

Input: Graph G
Output: The path set $P^{>u}(u,v)$ $(\forall u,v \in V)$
$L \leftarrow \emptyset$
for $i = 1,2,3,\ldots,n \in V$ **do**
  $L \leftarrow L \cup$ INDEX(i,R)
Return L

Hence, we can roughly estimate the time complexity of the whole algorithm. Suppose l is the average size of $L(v)$ $(\forall v \in V)$, we need to visit $O(lN)$ nodes in total. As in the analysis for pruned searching [33], [34], we assume in each node we have $O(\frac{M}{N})$ edges and each PRUNE needs $O(l)$ time. Thus, the total time complexity is $O(lM + lN(logN + l))$, where l is a pretty stable integer with our pruning strategy, much smaller than 300 in our experiment with real-world topologies.

V. SHORTEST GUARANTEED-COST PATH COMPUTATION

In this section we propose and characterize a solution to compute shortest guaranteed-cost paths.

A. Query guaranteed cost

We define the guaranteed cost $\eta(P(s,t))$ for a path $P(s,t)$ as $max\{\sigma(P'^{(s,t)}(s,n_i)) + \sigma(R(P'^{(s,t)}(n_i,t)))\}, \forall n_i \in P(s,t)$. Naïvely implementing it for all node pairs can lead a performance deficiency since the time complexity is $O(N^3)$. Therefore, [28] proposed a Dijkstra-like algorithm to compute the all-pair shortest guaranteed cost in $O(MN + N^2 \log N)$ by using Equation (6).

Theorem 1. $\forall s,t \in V$, $e_{s,n_1} \in P(s,t)$ we have

$$\eta(P(s,t)) = max \begin{cases} \sigma(e_{s,n_1}) + \eta(P(n_1,t)), \\ \sigma(R(P(s,t))). \end{cases} \quad (6)$$

There is a simple proof of Theorem 1 in [28]. Intuitively, Equation (6) explains the fact that the worst recovery case can only occur in the first edge of $P(s,t)$ or in the remaining edges of $P(s,t)$. Therefore, we can divide the computation of guaranteed cost into several sub-problems since the problem satisfies the *optimal substructure*. It is similar to the shortest path problem that also satisfies the *optimal substructure*. Thus, we can merge two sub-shortest guaranteed paths into one shortest guaranteed-cost path by Equation (7).

B. Indexing the shortest guaranteed-cost path

We define $Gp(s,t)$ as the shortest guaranteed-cost path that has the shortest guaranteed cost from s to t, and we define $SP(s,t)$ is the shortest path that has the lowest cost from s to t. We attempt to utilize the *2-hop cover* in indexing the shortest guaranteed path. Before detailing our algorithm design, we first prove that two sub-shortest guaranteed paths can merge into one shortest guaranteed-cost path.

Theorem 2. $\forall s,t \in V$, $\forall n_i \in Gp(s,t)$ we have

$$\eta(Gp(s,t)) = max \begin{cases} \eta(Gp(s,n_i)) + \sigma(SP(n_i,t)), \\ \sigma(SP(s,n_i)) + \eta(Gp(n_i,t)). \end{cases} \quad (7)$$

Proof 2. Suppose $P(s,t) = \{s, n_1, n_2, \ldots, n_m, t\}$, we decompose Equation (6), we have,

$$\begin{aligned} &\eta(P(s,t)) \\ &= max\{\sigma(R(P(s,t))), \sigma(e_{s,n_1}) + \eta(P'^{(s,t)}(n_1,t))\} \\ &= max \begin{cases} \sigma(R(P(s,t))), \\ \sigma(P'^{(s,t)}(s,n_i)) + \eta(P'^{(s,t)}(n_i,t)), \\ \sigma(P'^{(s,t)}(s,n_j)) + \sigma(R(P'^{(s,t)}(n_j,t))). \quad (\forall j < i) \end{cases} \end{aligned} \quad (8)$$

From the definition of $Gp(s,t)$, $\forall n_i \in P(s,t)$ we have

$$\begin{aligned} &\eta(Gp(s,t)) = min\{\eta(P(s,t)\} \\ &= min\{max\{\sigma(P'^{(s,t)}(s,n_i)) + \sigma(R(P'^{(s,t)}(n_i,t)))\}\}. \end{aligned} \quad (9)$$

We assume that n_β makes $\sigma(P'^{(s,t)}(s,n_\beta)) + \sigma(R(P'^{(s,t)}(n_\beta,t)))$ the maximum value (we can treat $\sigma(R(P(s,t)))$ as a special case with $n_\beta = s$), then from Equation (8) (9) we have

$$\begin{aligned} &\eta(Gp(s,t)) = min\{\sigma(P'^{(s,t)}(s,n_\beta)) + \sigma(R(P'^{(s,t)}(n_\beta,t)))\} \\ &= \sigma(SP(s,n_\beta)) + \eta(Gp(n_\beta,t)) \\ &= max\{\sigma(SP(s,n_i)) + \eta(Gp(n_i,t))\}. \quad (\forall i < \beta) \end{aligned}$$

In the same way, we have

$$\eta(Gp(s,t)) = max\{\eta(Gp(s,n_i)) + \sigma(SP(n_i,t))\}. \quad (\forall i > \beta)$$

Thus, $\forall n_i \in Gp(s,t)$, we have

$$\begin{aligned} \eta(Gp(s,t)) &= max \begin{cases} \eta(Gp(s,n_i)) + \sigma(SP(n_i,t)), & (\forall i > \beta) \\ \sigma(SP(s,n_i)) + \eta(Gp(n_i,t)). & (\forall i < \beta) \end{cases} \\ &= max \begin{cases} \eta(Gp(s,n_i)) + \sigma(SP(n_i,t)), \\ \sigma(SP(s,n_i)) + \eta(Gp(n_i,t)). \end{cases} \end{aligned}$$

□

C. Pruning strategy and pruned searching algorithm

We simply modify the existing Algorithm 1 to index the shortest guaranteed-cost path. We denote the guaranteed-cost path from u to v by $Gp^{>u}(u,v)$, of which all internal nodes are larger than u. In Algorithm 1, we use Equation (6) to replace $w.cost \leftarrow h.cost + \sigma(e_{h,w})$ with $w.cost \leftarrow max\{h.cost + \sigma(e_{h,w}, Query(w,u,e_{h,w})\}$ for expanding the new node in indexing. We use Equation (7) to implement the PRUNE and QUERY methods for a given pair of s and t to prune or query a shortest guaranteed-cost path from s to t. We prune the guaranteed-cost path whose cost is higher than the existing guaranteed-cost paths with the same s and t. We roughly estimate the time complexity. Similar to the definition of $L(v)$, we define $R(v)$ as the path set for all guaranteed-cost path whose destination is v. Thus, the total time complexity is $O(rM + rN(logN + r))$ where r is the average size of $R(v)$.

VI. EVALUATION

This section has two parts. The first part evaluates the performance of our algorithm with a variety of datasets. In the second part, we propose a prototype with Mininet and Open-vSwitch (OvS) to show the differences of the link recovery time between our solution and the practical implementation.

A. The Environment of Performance Evaluation

We first evaluate our algorithms through experiments on a Linux server with Intel i7-6700@4GHz and 32GB DDR4 memory. We use C++ to implement our algorithm, and make it publicly available at GitHub repository [50]. Since the efficiency of compiler optimization may vary based on different CPUs, we decide not to turn on the compiler optimization option in g++ to allow for a fair comparison. Thus, we use g++ to compile without any optimization. We use 2-byte integer (UNSIGNED SHORT) to represent the node ID and the cost. We evaluate the average query time by measuring $1,000,000$ random queries.

Datasets: To confirm that our methods are robust and scalable enough, we evaluate our algorithms on various real-world networks [35], [36]: three from carrier backbones, seven from AS networks. Although TOPO8 and TOPO9 have more than $20,000$ nodes, which are not common for SDN, we still use these topologies to test the scalability of our algorithm. We treat all topologies as undirected graphs. Since the original topology does not have cost information, we assign cost on each edge by a gamma distribution (from measurement results [51]) with $\alpha = 0.2463, \lambda = 55.9280$, and the scale of cost ranges from 1 to 524. We give the parameter of these networks in TABLE III.

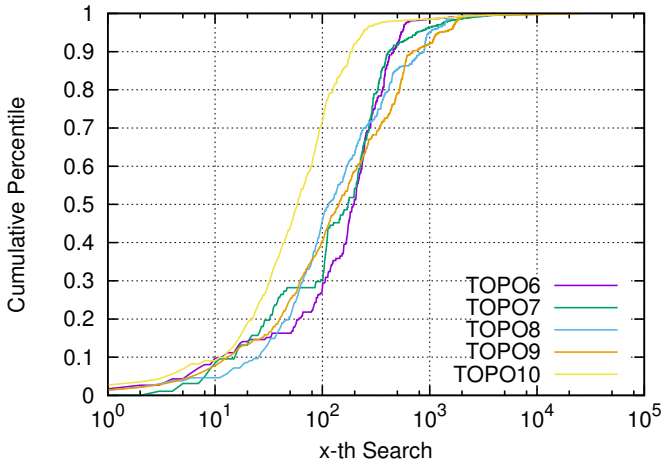
Since the degree of nodes can influence l, which is associated with the performance of our algorithms, we measure the degree distribution of the networks. These networks comply with the power-law distribution (80-20 rule), which means more than 80% degrees of vertexes are smaller than 2. Also, we measure the distribution of path cost on all-pairs nodes. We found 80% to 90% costs are smaller than 100.

B. Performance of computing the shortest recovery path

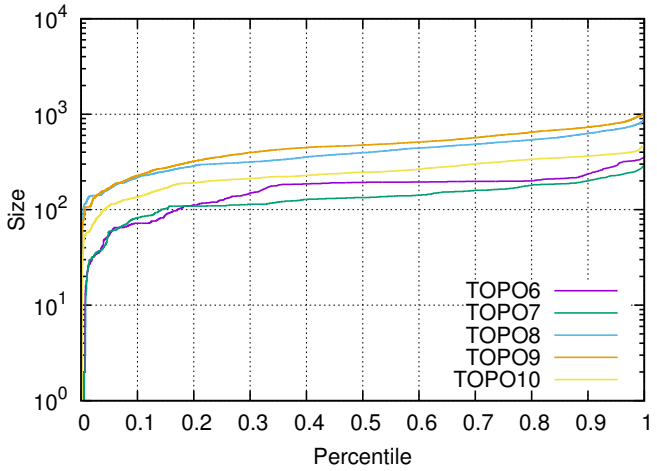
Firstly, we show the performance of computing the shortest recovery path on these real-world topologies in TABLE IV. IT indicates indexing times; MS indicates memory size, which is the memory usage of storing L and topology; QT indicates average querying time. CT indicates the computation time, which is composed of indexing time and querying time. We also evaluate the performance of two existing methods. One is optimized pre-computing [28], and the other is the naïve Dijkstra [27]. The latter is the algorithm that queries the shortest recovery path ($R(P(s,t))$) directly. We also implement these algorithms by using C++ and g++. Since the optimized pre-computing algorithm must compute all-pairs shortest recovery paths to get the correct result, CT of the optimized pre-computing indicates the time of computing recovery paths for all-pairs s and t. CT of the naïve Dijkstra indicates the time

TABLE III
DATASET

Dataset	Type	Date	$\|V\|$	$\|E\|$
TOPO1	BACKBONE Colt	201008	153	191
TOPO2	BACKBONE Cogent	201008	197	245
TOPO3	BACKBONE Kentucky	201008	754	889
TOPO4	AS IPv6	201605	1,163	1,274
TOPO5	AS IPv6	201605	2,439	2,797
TOPO6	AS IPv6	201605	4,079	4,955
TOPO7	AS IPv6	201605	5,340	6,572
TOPO8	AS IPv4	201601	21,394	42,622
TOPO9	AS IPv4	201602	25,150	50,874
TOPO10	Skitter	200802	7,292	15,258



(a) Cumulative distribution of the number of paths adding to L in each INDEX.



(b) Distribution of $|L(v)|$ ($\forall v \in V$)

Fig. 7. Characteristics of L in maximum five topologies.

of querying all-pairs nodes for shortest recovery paths in the graph. SR indicates the speedup ratio, which is computed by dividing the computation time of optimized pre-computing or naïve Dijkstra with the computation time in our algorithm. We estimate CT of TOPO8, TOPO9 by adding the indexing time with the querying multiply the number of nodes need to be protected, as the computation time for all-pairs protection is longer than half an hour, and protecting more than $20k$ nodes is not a common scenario in real-world. 'NA' indicates that the naïve Dijkstra and the optimized pre-computing cannot complete the computation within half an hour.

1) Indexing time and computation time: From TABLE IV we can see that the indexing time (Algorithm 2) is pretty small, showing the algorithm to be more scalable than others. Although the optimized pre-computing algorithm reduces a magnitude of the time complexity from naïve Dijkstra, the computation time in pruned searching is far smaller than the computation time in others. The computation time of our algorithm is longer than the optimized pre-computing only in TOPO1, TOPO2 and TOPO3, while these topologies are small topologies. As the overhead, the indexing time costs most in

TABLE IV
PERFORMANCE COMPARISON IN COMPUTING SHORTEST RECOVERY PATH
(IT: INDEXING TIME; MS: MEMORY USAGE; QT: QUERYING TIME, FOR A GIVEN PAIR OF s AND t;
CT: COMPUTATION TIME, FOR ALL-PAIR; SR: SPEEDUP RATIO)

Dataset	Pruned Searching					Optimized Pre-computing [28]			Naïve Dijkstra [27]	
10	IT	MS	QT	CT	l	CT	SR	MS	CT	SR
TOPO1	0.059s	15 MB	15μs	0.172s	15	0.138s	0.80x	15 MB	1.451s	8x
TOPO2	0.081s	32 MB	15μs	0.333s	53	0.246s	0.74x	22 MB	3.338s	10x
TOPO3	0.614s	817 MB	19μs	8.785s	86	8.716s	0.99x	97 MB	180.220s	20x
TOPO4	0.144s	157 MB	15μs	2.785s	20	6.248s	2.24x	104 MB	227.231s	81x
TOPO5	0.549s	230 MB	15μs	24.985s	32	77.726s	3.11x	243 MB	NA	NA
TOPO6	3.366s	1.0 GB	18μs	209.815s	74	494.182s	2.36x	1.4 GB	NA	NA
TOPO7	3.647s	1.8 GB	18μs	373.142s	66	721.314s	1.93x	3.5 GB	NA	NA
TOPO8	211.483s	11.1 GB	76μs	35023.174s	243	NA	NA	NA	NA	NA
TOPO9	258.017s	17.5 GB	83μs	52878.942s	277	NA	NA	NA	NA	NA
TOPO10	21.801s	5.6 GB	51μs	2736.015s	158	NA	NA	NA	NA	NA

TOPO1 and TOPO2.

2) Querying time: As we have mentioned above, the querying time depends on l. The query algorithm can query the result in microsecond level since all l is smaller than 10^3, and the longest QT is 83 μs. Although the optimized pre-computing algorithm does not utilize any time in a query, the computation time is too long to counteract the advantages. Moreover, our experiments show that the querying time does not increase rapidly with the increasing scale of the network.

3) Memory size: Our algorithm benefits from the *2-hop cover* policy and uses memory space to store intermediate results. We can see the memory usage in TOPO3 (817 MB) is larger than TOPO2 and TOPO4 since we have a larger l (86). The result is evidenced by the observation that there are many linear topology subgraphs in TOPO3, which makes l larger. We think the memory usage is sufficient for current commodity server that usually has more than $32GB$ memory. Also with the rapid development of modern memory technology, it should be easy to meet the memory space requirement.

Nevertheless, we still need to reduce the memory size for further improving scalability, which will be studied in future work.

C. Analysis

We analyze the reasons why our algorithm of computing the shortest recovery path is more efficient than other algorithms. We analyze some important characteristics of our algorithm in maximum five topologies.

1) Pruned searching: Fig. 7(a) shows the cumulative distribution of the number of paths added to L. From Fig. 7(a) we can see that most paths are added to L before calling INDEX 1,000 times (the second line of Algorithm 2), which means most of the computed recovery paths are unnecessary (pruned by PRUNE) after calling 1,000 times INDEX.

TABLE V
PERFORMANCE COMPARISON IN COMPUTING SHORTEST GUARANTEED-COST PATH
(IT: INDEXING TIME; MAXQT: MAXIMUM QUERYING TIME, FOR A GIVEN PAIR OF s AND t)

Dataset	Pruned Searching		Naïve algorithm [28]
5	IT	MaxQT	QT
TOPO6	0.468s	18μs	102ms
TOPO7	0.226s	16μs	154ms
TOPO8	8.355s	15μs	1.194s
TOPO9	4.739s	15μs	2.225s
TOPO10	2.479s	16μs	360ms

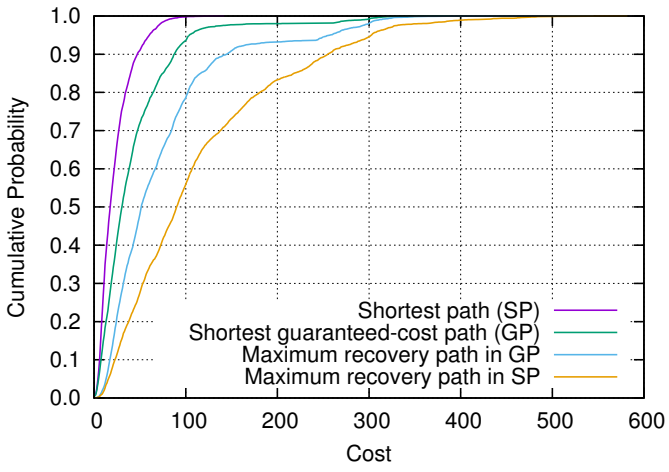
2) Size of L: Fig. 7(b) shows the distribution of $|L(v)|$ ($\forall v \in V$) with the increasing order of sizes. From Fig. 7(b) we can see that $\forall v \in V$, $|L(v)|$ are not much different among all nodes, and only a few nodes have a larger size, which means the querying time (related to the time complexity of QUERY and PRUNE) is pretty stable.

D. Performance of computing shortest guaranteed-cost path

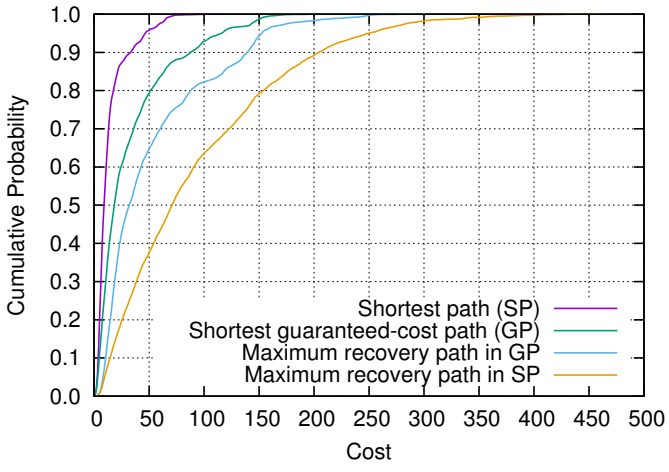
With the knowledge of all-pair shortest recovery path, we give the results of performance and characteristics evaluation in some large-scale topologies (TOPO6, TOPO7, TOPO8, TOPO9 and TOPO10). Large-scale topologies can provide more paths to analyze the property of shortest guaranteed-cost path. We give IT, maximum QT (most of the QT are pretty small because r are small) in TABLE V.

As we have mentioned, r is the average size of $R(v)$, which has a similar definition to the l and $L(v)$, indicating

the efficiency of the pruning strategy. In our evaluation, we find that r are 1 since most nodes in these networks do not have an alternative recovery path to reach other nodes (i.e., the degree of most nodes is 1). For example, a node with only one edge means it cannot find another path if the only edge failed. Thus, $|R|$ of these nodes are not larger than 1. On the other hand, if plenty of nodes having a large degree are available in the network, the IT will be smaller. The reason is that the node with a large degree (most paths must pass through these central vertices) can significantly increase the probability of pruning in pruned searching [33].



(a) Cumulative probability of the cost of the shortest path, the shortest guaranteed-cost path and the maximum recovery path in TOPO7.



(b) Cumulative probability of the cost of the shortest path, the shortest guaranteed-cost path and the maximum recovery path in TOPO10.

Fig. 8. The characteristic of the shortest guaranteed-cost path

E. The characteristic of the shortest guaranteed-cost path

We use TOPO7 and TOPO10 as examples to analyze the characteristic of the shortest guaranteed-cost path. We evaluate the characteristic of the shortest guaranteed-cost path by comparing the path cost with the shortest path, the shortest guaranteed-cost path, the maximum recovery path of the shortest path and the maximum recovery path of the shortest guaranteed-cost path.

We define the primary path as the path that packets pass through by default. From Fig. 8(a)(b), we can see that choosing the shortest guaranteed-cost path as the primary path can decrease the maximum cost of the recovery path notably. In Fig. 8(a), All recovery paths are shorter than 300 if we choose the shortest guaranteed-cost path as the primary path. Moreover, 80% of recovery paths are shorter than 100, which is more than 60% if we choose the shortest path as the primary path. Meanwhile, from Fig. 8(b) we can see that the maximum cost of recovery path decreases 150 cost (from 400 to 250) with only increasing the 50 cost (from 100 to 150) of the primary path slightly. Obviously, the shortest guaranteed-cost path can reduce the cost of the recovery path significantly.

F. Prototype Implementation and Evaluation

As mentioned, we have developed a prototype to validate the feasibility of our algorithm in real system. The prototype is based on Mininet and Open vSwitch (v 2.4.0-1). Note that we use some specific feature as the *Group Table* one, only available in OpenFlow versions later than 1.1.

Group Table: In order to implement Fast Reroute in a standard OpenFlow switch, we need to utilize *Group Table* feature. From the OpenFlow specification we know that *Group Table* enables OpenFlow to represent additional methods for forwarding. *Group Table* supports several forwarding modes, but the most important mode for us is the *fast failover* one. With the *fast failover* mode, OpenFlow switch can forward packet based on the port status (e.g., the port is up or down). In order to utilize the *fast failover* mode, we need to pass-through a pair of parameters: the watch port p and the action q; if p is up, the switch will perform action q. A group table can have several pairs of parameters, and each pair of parameters has its own priority. When a packet is matched, the OpenFlow switch will check the status of the watch port p_1 in the pair of parameters with the highest priority; if p_1 is live, then perform action q_1, otherwise the switch will check the status of the watch port $p_2, p_3 \dots p_n$ until a live watch port is found. If there is no live watch port, the switch will drop the packet. In our scenario, only two pairs of parameters are necessary in most cases. The first pair of parameters p_1 needs to be set as the port for forwarding packet with the shortest path (i.e., the primary path), and the action q_1 needs to be set to *forwarding to* p_1. The second pair of parameters has a lower priority. Note that p_2 needs to be set as the port for forwarding packet with the recovery path, and the action q_2 needs to be set to *forwarding to* p_2.

Fast Reroute algorithm in the controller: Group tables and entries need to be computed in the controller by our algorithm when the topology changes, and then be deployed into the switch. Currently, most controllers support a user defined path computation module, such as POX has FORWARDING.L2_MULTI module, and ONOS has GETPATHS() method to compute the path. Practically, in order to increase the flexibility, we can provide a recovery path computation server and modify the path computation module in different controllers as the client. The controller can use this client to get recovery path from the path computation server, and deploy them into the OpenFlow switch.

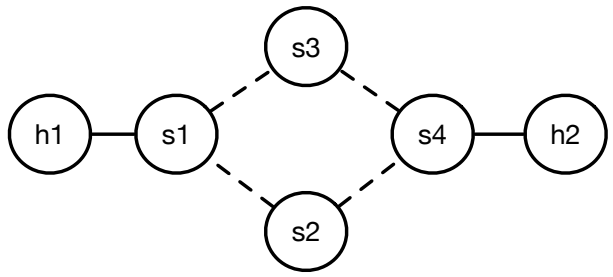


Fig. 9. A simple topology for prototype evaluation. We use *knock* to continuously send packets from $h1$ to $h2$. We random shut down the links between $s1$, $s2$, $s3$ and $s4$, and then measure the recovery time.

Topology and evaluation tools: We use Mininet to emulate our topology. In this evaluation, we use a simple topology (shown in Fig. 9) to test whether our system can work or not. Since Mininet can support a variety of topology formats such as 'GML'/'GraphML' by utilizing the *iGraph* library, we can easily change the topology if it needs it. We also use the Linux command *knock* to periodically send packets from the source to the destination.

G. Recovery Time Evaluation Results

We set the source at $h1$, and the destination at $h2$. *knock* in $h1$ will continuously trigger packets to $h2$. The server in $h2$ will print the reception time of packets. We can get recovery time by finding the first reception time after the link failure. We compute the recovery path for each link, and deploy them into switches. Then, we randomly shut down the link in mininet, and evaluate how much time is needed to recovery the communication. We evaluate the time with three recovery methods: (i) the recovery time without any Fast Reroute feature, the controller will check the connectivity periodically by LLDP; (ii) the recovery time that is the sum of switch reactively reports a failure to the controller, the controller uses the Naïve Dijkstra to compute the path, and deploys the new path into the switch; (iii) the recovery time with our Fast Reroute feature. In the third time, the switch does not need to connect with the controller. We use POX as controller to test case (i), use Ryu to test remaining cases since POX does not support OpenFlow 1.3 (Group Table feature). We compute recovery paths and deploy them by the controller.

TABLE VI
FAILURE RECOVERY TIME

	LLDP	Naïve Dijkstra	Fast Reroute
Time used	$9.235s$	$30.610ms$	$1.000ms$

From TABLE VI we can see that, since our solution does not need connect to the controller, the recovery time of Fast Reroute is the shortest.

VII. CONCLUSION

In this paper, we mainly focus on the performance bottleneck in recovery path computation for SDN fast rerouting. To accelerate the computation, we design two algorithms. First, we propose a fast algorithm for quickly indexing shortest recovery paths in a large-scale SDN. Second, we extend this algorithm to indexing the shortest guaranteed-cost path that has the minimum cost (delay) of the recovery path. We also prove the graph-theoretic correctness of the two algorithms. Moreover, we have design a prototype to show how to deploy our algorithm in the OpenFlow network. The evaluation results show that our algorithm can be about $8 \sim 81$ times faster than the practical implementation, 3.11 times faster than the theoretical legacy behavior in a $2,439$-node network. Furthermore, by analyzing the shortest guaranteed-cost path shape, the results show that the it can significantly reduce the cost of the recovery path.

REFERENCES

[1] S. Jain, A. Kumar, S. Mandal *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.
[2] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.
[3] H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey," *Elsevier Comput. Networks*, vol. 81, pp. 79–95, 2015.
[4] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," vol. 41, no. 4, pp. 350–361, 2011.
[5] S. B. C. C. Y. G. A. Markopoulou, G. Iannaccone and C. Diot, "Characterization of failures in an IP backbone," *IEEE/ACM Trans. Networking*, vol. 16, no. 4, 2008.
[6] G. Rétvári, L. Csikor *et al.*, "Optimizing IGP link costs for improving IP-level resilience," in *Proc. IEEE DRCN*, 2011, pp. 62–69.
[7] K.-W. Kwong, L. Gao, R. Guérin, and Z.-L. Zhang, "On the feasibility and efficacy of protection routing in IP networks," *IEEE/ACM Trans. Networking*, vol. 19, no. 5, pp. 1543–1556, 2011.
[8] Q. Li, M. Xu, Q. Li, D. Wang, and Y. Cui, "IP fast reroute: Notvia with early decapsulation," in *Proc. IEEE GLOBECOM*, 2010, pp. 1–6.
[9] A. Raj and O. C. Ibe, "A survey of IP and multiprotocol label switching fast reroute schemes," *Elsevier Comput. Networks*, vol. 51, no. 8, pp. 1882–1907, 2007.
[10] R. Bifulco and A. Matsiuk, "Towards scalable sdn switches: Enabling faster flow table entries installation," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 343–344, 2015.
[11] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Commun. Maga.*, vol. 2, no. 51, pp. 136–141, 2013.
[12] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever, "SWIFT: Predictive Fast Reroute," in *Proc. ACM SIGCOMM*, 2017, pp. 460–473.
[13] P. Pan, G. Swallow, and A. Atlas. (2005) Fast reroute extensions to RSVP-TE for LSP tunnels. [Online]. Available: www.rfc-editor.org/rfc/rfc4090.txt
[14] M. Shand and S. Bryant. (2010) IP fast reroute framework. [Online]. Available: www.rfc-editor.org/rfc/rfc5714.txt
[15] A. Atlas and A. Zinin. (2008) Basic specification for IP fast reroute: Loop-free alternates. [Online]. Available: www.rfc-editor.org/rfc/rfc5286.txt
[16] B. Stephens, A. L. Cox, and S. Rixner, "Scalable Multi-Failure Fast Failover via Forwarding Table Compression," in *Proc. ACM SIGCOMM SOSR*. ACM, 2016.
[17] Y.-D. Lin, H.-Y. Teng, C.-R. Hsu, C.-C. Liao *et al.*, "Fast failover and switchover for link failures and congestion in software defined networks," in *Proc. IEEE ICC*, 2016, pp. 1–6.
[18] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proc. ACM SIGCOMM HotSDN*, 2014, pp. 121–126.
[19] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *Proc. ACM SIGCOMM HotSDN*, 2013, pp. 109–114.
[20] D. Cai, A. Wielosz, and S. Wei, "Evolve carrier Ethernet architecture with SDN and segment routing," in *Proc. IEEE WowMom*, 2014, pp. 1–6.

[21] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in *Proc. IEEE DRCN)*, 2015, pp. 25–32.

[22] C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sansò, "Fast failure detection and recovery in SDN with stateful data plane," *Wiley Int. J. Network Mana.*, 2016.

[23] A. Voellmy and J. Wang, "Scalable software defined network controllers," in *Proc. ACM SIGCOMM*, 2012, pp. 289–290.

[24] M. Karakus and A. Durresi, "A Survey: Control Plane Scalability Issues and Approaches in Software-Defined Networking (SDN)," *Elsevier Comput. Networks*, vol. 112, pp. 279–293, 2017.

[25] P. C. da Rocha Fonseca and E. S. Mota, "A Survey on Fault Management in Software-Defined Networks requirements," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2284–2321, 2017.

[26] D. Merling, W. Braun, and M. Menth, "Efficient data plane protection for SDN," in *Proc. IEEE Netsoft*, pp. 10–18.

[27] R. M. Ramos, M. Martinello, and C. E. Rothenberg, "SlickFlow: Resilient source routing in data center networks unlocked by OpenFlow," in *Proc. IEEE LCN*, 2013, pp. 606–613.

[28] A. Jarry, "Fast reroute paths algorithms," *Springer Telecomm. Systems*, vol. 52, no. 2, pp. 881–888, 2013.

[29] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proc. ACM SIGCOMM HotSDN*, 2014, pp. 1–6.

[30] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *Proc. IEEE WowMoM*, 2014, pp. 1–6.

[31] J. Hershberger and S. Suri, "Vickrey prices and shortest paths: What is an edge worth?" in *Proc. IEEE FOCS*, 2001, pp. 252–259.

[32] Z. Gotthilf and M. Lewenstein, "Improved algorithms for the k simple shortest paths and the replacement paths problems," *Elsevier Info. Proc. Lett.*, vol. 109, no. 7, pp. 352–355, 2009.

[33] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proc. ACM SIGMOD*, 2013, pp. 349–360.

[34] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida, "Efficient Top-k Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling," in *Proc. AAAI*, 2015, pp. 2–8.

[35] Topology zoo. [Online]. Available: http://www.topology-zoo.org

[36] Caida. [Online]. Available: www.caida.org

[37] V. Sharma. (2003) Framework for multi-protocol label switching (mpls)-based recovery. [Online]. Available: www.ietf.org/rfc/rfc3469.txt

[38] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. (2001) RSVP-TE: extensions to RSVP for LSP tunnels. [Online]. Available: www.rfc-editor.org/rfc/rfc3209.txt

[39] H. Kim, M. Schlansker, J. R. Santos *et al.*, "Coronet: Fault tolerance for software defined networks," in *Proc. IEEE ICNP*, 2012, pp. 1–2.

[40] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Proc. IEEE EWSDN*, 2014, pp. 61–66.

[41] N. Sahri and K. Okamura, "Fast failover mechanism for software defined networking: Openflow based," in *Proc. ACM CFI*, 2014, p. 16.

[42] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "Research challenges for traffic engineering in software defined networks," *IEEE Network*, vol. 30, no. 3, pp. 52–58, 2016.

[43] (2016) Intel i7 920 MIPS benchmark. [Online]. Available: www.cpu-world.com

[44] H. Yamanaka, E. Kawai, and S. Shimojo, "Scaling-up Flow Path Computation for a Large Number of Virtual SDN/NFV Infrastructures (DEMO)," in *Proc. ACM SIGCOMM SOSR*. ACM, 2015.

[45] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," *Elsevier Comput. Commun.*, vol. 36, no. 6, pp. 656–665, 2013.

[46] N. L. van Adrichem, F. Iqbal, and F. A. Kuipers, "Backup rules in software-defined networks," in *Proc. IEEE NFV-SDN*, 2016, pp. 179–185.

[47] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Elsevier J. Algorithm*, vol. 49, no. 1, pp. 114–152, 2003.

[48] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proc. ACM SIGKDD*, 2003, pp. 137–146.

[49] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 6.

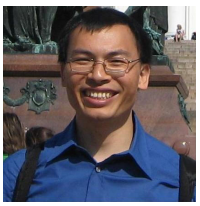
[50] Fast reroute source code. [Online]. Available: https://github.com/flyfox141/effirecovery

[51] H. Zhang, Y. Zhang, and Y. Liu, "Modeling Internet link delay based on measurement," in *Proc. IEEE ICECT*, 2009, pp. 420–424.

Kun Qiu received his B.Sc. Degree from Fudan University in 2013, and received his Ph.D. Degree from Fudan University in 2019. His research interests include computer network and computer architecture. He stayed at Universite Pierre et Marie Curie and University of Goettingen for half a year as a visiting student. He is a student member of IEEE ACM and CCF.

Jin Zhao received his B. Eng. Degree in computer communications from Nanjing University of Posts and Telecommunications, China, in 2001, and the Ph.D. Degree in computer science from Nanjing University, China, in 2006. He joined Fudan University in 2006. He stayed at University of Massachusetts Amherst for 1 year as a visiting scholar in 2014. His research interests include software defined networking, media streaming and network coding theory. He is a member of IEEE and ACM.

Xin Wang received his Bachelor Degree and the Master Degree from Xidian University, Xi'an, China, in 1994 and 1997 respectively, in Information Theory and Communications. He received the Ph.D. Degree from Shizuoka University, Japan in 2002, in Computer Science. Since 2002, he has been with the School of Computer Science at Fudan University, where he is currently a full professor. He is a member of IEEE and CCF.



Xiaoming Fu received his Ph.D. in Computer Science from Tsinghua University, China in 2000. He is a professor at the Georg-August-Universitaï Goïtingen. He has also held visiting positions at ETSI, University of Cambridge, Columbia University, Tsinghua University, and UCLA. He is a senior member of IEEE, member of ACM.

Stefano Secci is a Professor of networking at the Cnam, Paris, France. Previously he was associate professor at Sorbonne University-UPMC, Paris, France. He received the M.Sc. Degree in communications engineering from Politecnico di Milano, Milan, Italy, in 2005, and a dual Ph.D. Degree in computer science and networks from Politecnico di Milano and Telecom ParisTech, France, and held postdoc positions at NTNU, Norway, and GMU, USA. Webpage: http://cedric.cnam.fr/~seccis. He is a senior member of IEEE.