Sector: TCAM Space Aware Routing on SDN

Sai Qian Zhang, Qi Zhang, Ali Tizghadam, Byungchul Park, Hadi Bannazadeh, Raouf Boutaba and Alberto Leon-Garcia

***Abstract*—In Software Defined Networking (SDN), fine-grained control over individual flow can be achieved by installing appropriate forwarding rules in switches and routers. This allows the network to realize a wide variety of functionalities and objectives. But at the same time, this flexibility and versatility come at the expense of (1) a huge burden on the limited Ternary Content Addressable Memory (TCAM) space, and (2) limited scalability due to the large number of forwarding rules handled by the controller. To address these limitations, we present *Sector*, a switch memory-aware routing scheme that reduces TCAM space usage without introducing network congestion. We consider static and dynamic versions and propose corresponding solution algorithms. Experiments show our algorithms can reduce TCAM space usage and network control traffic by $20\% - 80\%$ compared with the benchmark algorithms on different network topologies.**

I. INTRODUCTION

Software Defined Networking (SDN) is an architecture that enables logically centralized control over distributed network resources. In SDN, a centralized controller makes forwarding decisions on behalf of the network forwarding elements (e.g. switches and routers) using a set of policies. Based on given high level design requirements, the source and the destination node of each flow is dictated by the *Endpoint Policy* and the flow path is decided by the *Routing Policy* [1]. For example, the shortest-path routing policy asks the network to forward packets along the shortest path between two nodes. Other routing polices, such as the ones that improve resource utilization, quality of service and energy usage have also been proposed in the literature [2,3,4]. These features make SDN an attractive approach for realizing a wide variety of networking features and functionalities.

Despite its benefits, however, implementing routing policies in SDN may require fine-grained control over flows, which can place a huge burden on switch memory space. In particular, the Ternary Content Addressable Memory (TCAM) is a special type of high speed memory that can search the entire memory space within a single clock cycle. However, it is also well known to have limited capacity and high power consumption [5]. The largest memory space on a TCAM chip is far less than that of Binary Content Addressable Memory (CAM). For example, HP ProCurve 5406zl TCAM switch hardware can support 1500 OpenFlow rules. As each host requires dozens of OpenFlow rules on average, a 5406zl switch can only support at most 150 users [6]. Moreover, TCAM is also energy-hungry. It consumes 30 times as much as the consumed energy of SRAM with the equal number of entries [7]. Given that the amount of power consumption is proportional to the number of used entries in TCAM, several research studies have focused on reducing the TCAM space consumption [1,5,8] in order to improve scalability and reduce energy consumption.

Another scalability problem arise in the centralized SDN controller. For every subtle change on the network topology or routing policy, the controller must deliver the control message to each network element that implements the policy. As the average flow size in both wide-area and data center networks is small (around 20 packets per flow [6]) and the inter-arrival rate of the flows in the high-performance network is extremely high (less than 30ms [6]), a huge workload is imposed on the controller as the network size grows. Since each switch typically has limited bandwidth on the path to the controller, and moderate rule insertion time, the high workload received by the controller often causes large rule installation overhead and low flow set up rate. In the modern data center networks, 1ms additional latency for the delay-sensitive flow can be intolerable [9]. Therefore, the limited flow set up rate can dramatically hurt the overall performance and the quality of service. It is important to reduce the interaction between control plane and data plane in order to achieve better network scalability and performance.

To address the issues of switch memory space limitation and scalability, recent work proposed to control flows collectively at an aggregated level. This allows the use of prefix aggregation and wild card rules to minimize the number of stored entries [1][8]. These works have focused on compressing the entries of each individual switch, while preserving the routing policy (i.e. without changing the forwarding paths) [5]. However, we find that in large networks, typically multiple candidate paths are available for routing each individual flow while still satisfying performance and business constraints. Therefore, if we can additionally control the flow forwarding paths, we achieve substantial gain in term of TCAM space savings and controller scalability. To this end, we present *Sector*, a routing scheme that minimizes TCAM space consumption in SDN networks without causing network congestion. *Sector* takes advantages of the large number of available forwarding paths and routes traffic in a way that improves network scalability and reliability. The main objectives of *Sector* are (1) Minimizing the switch memory space utilization given the end point connection request, and (2) Reducing control traffic by decreasing the interaction between the controller and network.

In this paper, we first introduce the TCAM space minimization problem and analyze its complexity. We then propose heuristic algorithms for both static and dynamic versions of the problem. Through experiments, we show our algorithms can reduce the TCAM space usage and network control traffic by $20\% - 80\%$ compared with the benchmark algorithms.

The rest of paper is organized as follows. Section II surveys related work. Section III provides a motivating example of the TCAM space minimization problem. Section IV presents the problem overview. Section V and VI presents our solutions for the static version of the problem. Section VII presents our solution for the dynamic version of the problem. After presenting experimental results and testbed implementation in Section VIII and IX, we conclude the paper in Section X.

II. Background and Related Work

OpenFlow is the most popular implementation of SDN [10]. An OpenFlow table entry can be represented by a triplet (M, P, A), where M is the matching field used to match the packet, P is the matching precedence of the entry and A is the action field which contains operations on the matched packet. The matching field usually includes source and destination IP addresses, MAC addresses and input port number. The action field includes operations such as forwarding the packet to a output port or modifying the packet header. Upon receiving a packet, the switch searches for the rule with the highest priority that matches the packet, then executes corresponding actions defined by that rule. OpenFlow also supports wildcards in the *input port* as well as subnet masks in IP and MAC addresses [10], for instance, $01**$ in the address field stands for 0100, 0101, 0110 and 0111.

There are several studies on minimizing TCAM space using subnet masks and wildcard rules. However, prior work has focused on compressing the entries of a single switch, while preserving the routing policy [5]. One Big Switch [1] and Palette [8] decompose network access policies into small pieces and then distribute them using less TCAM space. Moshref et al. [11] designs routing algorithms to distribute access policies across intermediate switches with minimum switch memory consumption in a datacenter network. Rami et al [24] studies the effect of flow table size on the maximum number of flows supported. CacheFlow [25] develops a algorithm for placing rules in a TCAM with a limited space.

Scalability has been a key issue in SDN. DevoFlow [6] proposes a scalable SDN framework by using wildcard entries to reduce the control plane visibility on the microflows. However, it does not offer any quantitative analysis on how to use the wildcard to achieve optimal performance. DIFANE and Kandoo [12,13] propose efficient and scalable SDN frameworks which split the workload of the central controller to distributed authorized components. However, they do not address the problem of global visibility. In [14,15], the scalability issue is solved by using multiple independent controllers to consistently manage the network with minimal interaction, but they do not mention how these controllers are coordinated and the overhead brought by distributed control.

III. A Motivating Example

We provide a motivating example to demonstrate the benefit of Sector. The topology and port numbers between nodes are shown in Figure 1(a) and the end point policy is illustrated in Figure 2(a). Two source hosts with the IP address 000 and 001 send traffic to two destination hosts 100 and 101 respectively. We call a group of source and destination address a demand pair, therefore there are four demand pairs in this example. The bandwidth consumption of each demand pair

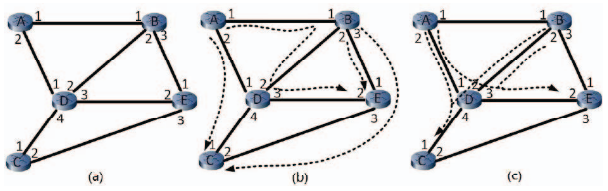
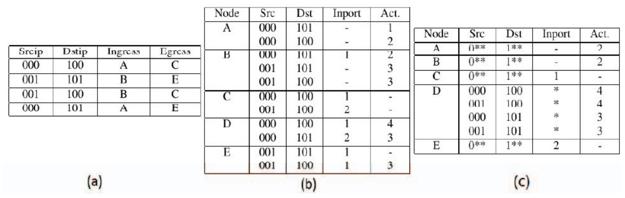


Fig. 1. Motivation example



(a)

Srcip	Dstip	Ingress	Egress
000	100	A	C
001	101	B	E
001	100	B	C
000	101	A	E

(b)

Node	Src	Dst	Inport	Act.
A	000	101	-	1
	000	100	-	2
B	000	101	1	2
	001	101	-	3
	001	100	-	3
C	000	100	1	-
	001	100	2	-
D	000	100	1	4
	000	101	2	3
E	001	101	1	-
	001	100	1	3

(c)

Node	Src	Dst	Inport	Act.
A	0**	1**	-	2
B	0**	1**	-	2
C	0**	1**	1	-
D	000	100	*	4
	001	100	*	4
	000	101	*	3
	001	101	*	3
E	0**	1**	2	-

Fig. 2. Example of flow tables

equals 1 and the capacity of each link is 10. Traditional traffic engineering (e.g. ECMP) spreads the flows evenly in the network to balance network link utilization, which gives one of the feasible solutions in Figure 1(b). The OpenFlow table of each switch is shown in the Figure 2(b). A total of 11 entries is installed. To set up these new flows, 11 additional control packets are sent from the controller, as at least one initial packet in each new flow is processed by the controller. In total $11 + 2 \times 4 = 19$ packets are transmitted between controller and the switches. By comparison, *Sector* produces the solution in Figure 1(c) and forwarding tables shown in Figure 2(c). Instead of routing the traffic of each demand pair respectively, *Sector* aggregates the flows and uses subnet masks to reduce the number of entries in each table. The maximum bandwidth consumption in the Sector solution is also 2, and 8 additional entries are installed on nodes $A - E$, which requires 8 control packets sent from controller. A total $8 + 2 \times 4 = 16$ packets are transmitted between controller and switches. This reduces TCAM space and control traffic by 27.2% and 15.8% respectively. From the above example, we draw the following conclusions: (1) If fine-grained control is not required on specific flows, TCAM space consumption and control traffic can be reduced by using subnet masks on the source and destination addresses to aggregate flow entries. (2) As the network size increases, the number of control packets to set up a flow is approximately equal to the number of entries installed in the TCAM (ignoring the initial packet of the flow send to the controller). So minimizing TCAM usage can also save control traffic indirectly. (3) Besides finding a path which minimizes TCAM consumption, the constraint on link capacity must also be guaranteed. For example, the two solutions above have the same maximum link utilization.

IV. Problem Overview

The design objective of Sector is to minimize the total number of OpenFlow entries installed in all the switches. To keep the problem generic, we assign a weight $w(v)$ to each switch v in the network, which is the cost of installing an additional rule in the switch. For the choice of $w(v)$, in the simplest case, we can set $w(v) = 1$ to achieve the goal of minimizing total number of forwarding entries in the switches.

Moreover, adjusting the value of $w(v)$ allows us to model other objectives. For instance, since power consumption of a switch is linearly proportional to the TCAM space usage [5], by setting $w(v)$ to the average power consumption per rule for switch v, we can model the problem of minimizing total energy consumption in the network. The objective is therefore to minimize the total weighted cost, given a set of demand pairs and the constraint on link resource utilization. We call this the *TCAM Space Minimization Problem (TSMP)*.

TSMP is a rather complex problem to analyze and solve directly. To simplify our analysis, we divide TSMP into two sub problems: *Efficient Partitioning Problem (EPP)* and *Efficient Routing Problem (ERP)*. The EPP focuses on partitioning all the demand pairs into groups. We call these groups the *routing groups*. The source addresses and destination addresses in the same routing group have common prefixes. For example, the four demand pairs $[000, 100], [000, 101], [001, 100], [001, 101]$ in the Figure 2(a) form a routing group with prefix $0**$ and $1**$, where we use $[s_k, d_k]$ to represent the demand pair. We can use the addresses with subnet mask $s_u = 0**$ and $d_u = 1**$ to represent all the source addresses and destination addresses in the routing group u. When partitioning is complete, for each routing group there will be a ERP, where we route all demand pairs in that routing group to minimize total TCAM space usage.

In the next two sections we first discuss our algorithm for ERP, and then the solution for EPP, which relies on the solution algorithm for ERP to make partitioning decisions.

V. Efficient Routing problem

The goal of ERP is to connect each demand pair for a given routing group that consume minimum weighted sum of switch memory space while satisfy the load balancing on links. Formally, we model the network as a graph $G = (V, E)$, where each node $v \in V$ represents an OpenFlow switch and each switch v is assigned a cost $w(v)$ on per rule inserted. Without loss of generality, we assume each flow entry in the flow table can be represented by a 4-tuple (s, i, d, j), where s, i, d constitute the matching field: s, d represent the source and destination address information such as source/destination IP/MAC address, i is the input port number of the switch where the packet comes in. j is the output port number of the switch where the packet is directed to, which constitutes the action field of the OpenFlow entry. We neglect rule priority temporarily and consider it later.

Let U denote the set of routing groups and $K_u (u \in U)$ denote the set of demand pairs in u. Let s_k and d_k denote the source and destination addresses of demand pair k. We use a 4-tuple (s_u, i, d_u, j) to represent the OpenFlow rule installed for the routing group $u \in U$, where s_u and d_u are the source and destination addresses with the subnet masks respectively. Table I provides a quick glossary of definitions. Let $\pi(v)$ be the set of port numbers of switch v, we make the port number equals to the label of the links that the port connects to (Figure 3). Then denote $p(v) = \{(x, y) : x \in \pi(v), y \in \pi(v)\}$ as the set of port pairs of switch v. For example $\pi(A)$ in Figure 3 is $\{1, 2, 4\}$ and $p(A) = \{(1, 2), (2, 1), (1, 4), (4, 1), (2, 4), (4, 2), (1, 1), (2, 2), (4, 4)\}$. Let $y_{ij} \in \{0, 1\}$ represent whether a 4-tuple is installed to direct the flow of routing group u from input port i to output port j. Let $x_{ijk} \in \{0, 1\}$ denote whether a 4-tuple entry is installed to directed traffic of a demand pair $k \in K_u$ from input port i to output port j. Let $a(v)$ denote the total number of rules installed on switch v. Our goal is to minimize the total weighted sum of rules installed in the switches:

$$\underset{x_{ijk}, y_{ij} \in \{0,1\}}{\text{minimize}} \sum_{v \in V} w(v) a(v) \tag{1}$$

where $a(v)$ represents the number of 4-tuples (s_u, i, d_u, j) installed in v. To compute $a(v)$, note that for the same switch v and same routing groups, three conditions may happen:

1. No 4-tuple (s_u, i, d_u, j) needs to be installed on v. That is, $\sum_{j \in \pi(v)} \mu(\sum_{i \in \pi(v)} y_{ij}) = 0$ and therefore $a(v) = 0$. Where μ is the step function, $\mu(x) = 0$ if $x \leq 0$ and $\mu(x) = 1$ if $x > 0$.

2. All the flows installed on v are forwarded to one output port, i.e., $\sum_{j \in \pi(v)} \mu(\sum_{i \in \pi(v)} y_{ij}) = 1$. One entry $(s_u, *, d_u, j)$ is enough to direct the flows of K_u, with s_u and d_u in the address field and wildcard in the input port field. therefore $a(v) = 1$.

TABLE I. Definitions of parameters

Name	Description
G	A network topology $G = (V, E)$
V	Set of nodes in G
E	Set of links in G
S	Set of addresses of source hosts
D	Set of addresses of destination hosts
U	denote the set of routing groups
K	Set of demand pairs
K_u	Set of demand pairs in the routing group $u \in U$
m	Number of bits in the source address and destination address
$size(u)$	The number of demand pairs in routing group u
L	Maximum number of demand pairs in each routing group
s_k	the source address of demand pair k
d_k	the destination address of demand pair k
s_u	The source address of routing group u with the subnet mask
d_u	The destination addr. of routing group u with the subnet mask
$a(v)$	Number of OpenFlow rules installed on switch v
$w(v)$	Cost of inserting a single OpenFlow rule in switch v
r_v	The TCAM space capacity of switch v
$\pi(v)$	Set of port numbers associated with switch v
$p(v)$	Set of port number pairs of switch v
β	Threshold of link utilization rate
B_k	The bandwidth consumption of $k \in K$
C_e	C_e the capacity of each link $e \in E$
x_{ijk}	A binary variable, $x_{ijk} = 1$ if an 4-tuple (s_u, i, d_u, j) is installed to direct traffic of demand pair k from port i to port j, $x_{ijk} = 0$ otherwise
y_{ij}	A binary variable, $y_{ij} = 1$ if a 4-tuple (s_u, i, d_u, j) is installed to direct the flow of $s_u \in S$ from port i to port j and $y_{ij} = 0$ otherwise
l_{ek}	A binary variable, $l_{ek} = 1$ denotes edge $e \in E$ is used to direct the flow of demand pair k

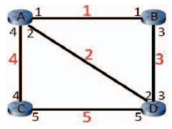
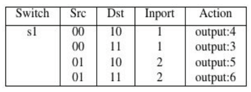


Fig. 3. Labelling port example



Switch	Src	Dst	Inport	Action
s1	00	10	1	output:4
	00	11	1	output:3
	01	10	2	output:5
	01	11	2	output:6

Fig. 4. Flow Table of s1

3. All the flows installed on v are forwarded to more than one output port. That is, $\sum_{j\in\pi(v)}\mu(\sum_{i\in\pi(v)}y_{ij}) > 1$, therefore, the source and destination fields must be fully specified to differentiate each flow and so that the flows can be directed to corresponding output ports. Hence the total number of entries installed is $\sum_{i\in\pi(v)}\sum_{j\in\pi(v)}\sum_{k\in K_u}x_{ijk}$, which is the number of demand pairs whose flows traverse through v. This can be illustrated by the following example: Assume a set of rules $\{(00,1,10,4),(01,2,10,5),(00,1,11,3),(01,2,11,6)\}$ is installed on $s1$. The flow table of $s1$ shown in Figure 4. As the table shows, the source and destination address must be fully specified so that each flow can be identified by the intermediate switch to direct to its corresponding output port.

Combining these 3 cases, $a(v)$ can be defined as follows:

$$a(v) = \begin{cases} 0 & \text{if } \sum\limits_{j\in\pi(v)}\mu(\sum\limits_{i\in\pi(v)}y_{ij}) = 0 \\ 1 & \text{if } \sum\limits_{j\in\pi(v)}\mu(\sum\limits_{i\in\pi(v)}y_{ij}) = 1 \\ \sum\limits_{i\in\pi(v)}\sum\limits_{j\in\pi(v)}\sum\limits_{k\in K_u}x_{ijk} & \text{if } \sum\limits_{j\in\pi(v)}\mu(\sum\limits_{i\in\pi(v)}y_{ij}) > 1 \end{cases}$$

We also have to make sure that the number of rules installed in each switch does not exceed its TCAM space capacity, let r_v be the capacity of switch v, we then have:

$$a(v) \le r_v \quad \forall v \in V \tag{2}$$

Next, we relate x_{ijk} to y_{ij}. Equation (3) ensures that 4-tuple rule (s_u, i, d_u, j) is installed if any flow of demand pair k is sent from input port i to output port j:

$$\sum_{k:k\in K_u} x_{ijk} \le y_{ij} \quad \forall (i,j)\in p(v), v\in V \tag{3}$$

Next we build the path between each source host to the destination host. Let $l_{ek}\in\{0,1\}$ denote if edge $e\in E$ is used to direct the flow of demand pair k. Define $Q_k = \{Q_k \subseteq V : s_k\in Q_k, d_k\notin Q_k\}$ $(\forall k\in K)$ and define $\pi(Q_k)$ the set of edges in the *cut* defined by Q_k, that is the set of edges in G which have ingress node in the set Q_k. Then we have:

$$\sum_{e:e\in\pi(Q_k)} l_{ek} \ge 1 \quad \forall k\in K_u \tag{4}$$

By *max-flow/min-cut* theorem, equation (4) ensures there exists at least one path between s_k and d_k [16]. Next the following equations make sure OpenFlow entries are installed to direct the flow to each used link:

$$l_{ek} \le \sum_{v\in V}\sum_{i:(i,e)\in p(v)} x_{iek} \le 1 \quad \forall e\in E, k\in K_u \tag{5}$$

$$l_{ek} \le \sum_{v\in V}\sum_{j:(e,j)\in p(v)} x_{ejk} \le 1 \quad \forall e\in E, k\in K_u \tag{6}$$

$$\sum_{(i,j)\in p(v)} x_{ijk} \le 1 \quad \forall k\in K_u, v\in V \tag{7}$$

Equations (5)(6)(7) ensure that if link e is used to direct the flow for k, then there exists exact one flow entry in the ingress switch of e to direct the flow of k to e and there exists one flow entry in the egress switch of e to accept the flow of k from link e. Finally, we have the performance guarantee on maximum bandwidth utilization rate for all the links. Define B_k the bandwidth consumption for the demand pair k, C_e the capacity of each link $e\in E$, and define β the threshold on link utilization rate. we have:

$$\sum_{k:k\in K} B_k l_{ek} \le \beta C_e \quad e\in E \tag{8}$$

The goal of ERP is to minimize objective function (1), subject to equations $(2)-(8)$.

Next we analyze the complexity of ERP, Theorem 1 shows the NP-completeness and inapproximability of the ERP. Theorem 2 shows that even without the load balancing guarantee (8), or with some other performance guarantee rather than (8), the ERP is still NP-hard and $(1-\epsilon)\ln|V|$ inapproximable for any $\epsilon > 0$.

Theorem 1. ERP is NP-complete and inapproximable.

Proof: The proof is based on reduction from the *3-partition problem*[1]. Consider the part of hierarchical tree topology in modern datacenter in Figure 5(a). Four source hosts inject packets to A, B, C, D, and the bandwidth consumption B_k of the traffic injected on A, B, C, D are b_1, b_2, b_3, b_4 respectively. The maximum usage on bandwidth βC_e of link (E,H),(G,H) and (F,H) equal $\frac{1}{3}(b_1+b_2+b_3+b_4)$. To satisfy (8), the flows from the four source nodes must be partitioned into three subsets with the same total amount of bandwidth $\frac{1}{3}(b_1+b_2+b_3+b_4)$. Therefore by knowing whether the problem is feasible or not, we know whether the set of numbers $\{b_1, b_2, b_3, b_4\}$ can be partitioned into three subsets with the equal sum of elements. Since the decision version of 3-partition problem is NP-complete, hence any polynomial-time approximation algorithm for this problem would solve the 3-partition problem in polynomial time, which is not possible unless $P = NP$. ■

Following the same arguments, we can also show that $TSMP$ is also NP-complete and inapproximable.

Theorem 2. Even without the link capacity constraints (i.e., equation (8)), ERP defined by $(1)-(7)$ is NP-hard, and there is no $(1-\epsilon)\ln|V|$-approximation algorithm for any $\epsilon > 0$, where $|V|$ is the number of nodes in G.

Proof: The proof is based on a reduction from the *set cover problem*[2]. Consider a multi-root hierarchical tree topology in Figure 5(b), each node on layer 3 does not fully connect to every node on layer 2 due to link failure. 4 source hosts forms a routing group, each connects with the switches A, B, C, D and send traffic to the core switch H. Assume r_v is large and the weight of all the switches on layer 3 and layer 1 is small, the objective functions (1) is equivalent to minimize the number of entries inserted on layer 2 switches.

Since each additional switch used on layer 2 to direct the flow from $A-D$ corresponds to an additional flow entry

[1]The partition problem is the task of deciding whether a set of positive integers can be partitioned into three subsets X, Y and Z such that the sums of the numbers in X, Y, Z are equal

[2]Given a set of elements {1,2,...,m}, and a set A of n sets whose union equals the element set. The set cover problem is to find the smallest subset of A whose union contains every single element

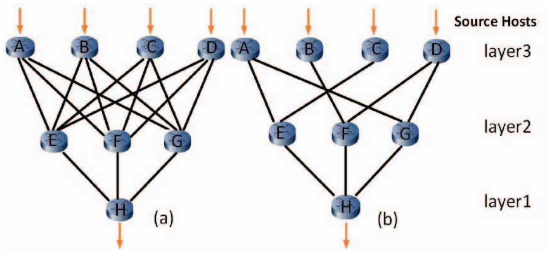
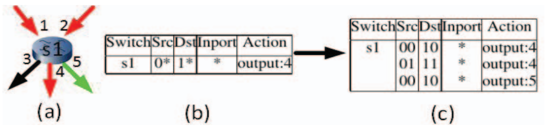


Fig. 5. Proof for Inapproximable



Switch	Src	Dst	Inport	Action
s1	0*	1*	*	output:4

Switch	Src	Dst	Inport	Action
s1	00	10	*	output:4
	01	11	*	output:4
	00	10	*	output:5

Fig. 6. Example on cost of link

inserted on that switch, minimizing number of entries on layer 2 switches is equivalent to minimizing the number of switches used on layer 2. Define the universal set $U = \{A, B, C, D\}$ which consists of all the layer 3 switches and assign a subset of U to each switch on layer 2, the subset for each switch on layer 2 consists of the switches on layer 3 that switch connects with. For example, the subset for $E = \{A, C\}$ and the subset for $F = \{B, D\}$. In order to make sure there is a path from $A - D$ to destination H, we need to ensure each switch on layer 3 connects to at least one switch on layer 2. Therefore, minimizing the number of additional flows inserted on layer 2 switches is equivalent to minimizing the number of layer 2 switches used to direct the flow, which is equivalent to minimizing the number of subsets used to cover the universal set U, which is the definition of set cover problem. Since the set cover problem is NP-hard and cannot be approximated with in a factor of $(1 - \epsilon) \ln n$ for any $\epsilon > 0$ (where n is the size of the set). The ERP is also NP-hard and $(1 - \epsilon) \ln |V|$ inapproximable for any $\epsilon > 0$. ■

Since ERP is both NP-complete and inapproximable, we propose a simple and efficient heuristic to solve ERP. Without loss of generality, given an undirected topology $G = (V, E)$ the graph can be made directed by replacing each undirected link e by two directed links e' with opposite directions, mark both directed links e' evolved from e. We define a new directed graph $G' = (V', E')$, and $in(e')(e' \in E')$ as the *ingress switch* (head) of e' and $out(e')(e' \in E')$ as the *egress switch* (tail) of e', an directed link e' is a link from its *egress switch* (tail) to its *ingress switch* (head). Define $C_{e'}(e' \in E')$ the capacity of the link e', which equals that of C_e, where e is the undirected link from which e' is created. We relate the cost of inserting rules on switches to the weight of the directed links of the switches. First, we provide the following definition:

Definition 1. Link e' is *ready* for routing group u if: 1. $out(e')$ contains a 4-tuple $(s_u, i, d_u, e'), i \in \pi(out(e'))$ or $(s_u, *, d_u, e')$. 2. $in(e')$ contains a 4-tuple $(s_u, e', d_u, j), j \in \pi(in(e'))$ or $(s_u, *, d_u, j)$.

That is, a link is *ready* for u if there already exists an entry on its ingress switch and egress switch to forward the flow onto this link. Next we calculate the cost of activating the links e' on switch $out(e')$. Let $t(v)(v \in V)$ be the number of demand pairs of u that v carries after the e' is activated. Define θ_v^u the number of egress links of v used to direct the traffic of demand pairs of u before e' is added. Then the cost of activating this

Algorithm 1 Incremental Routing Algorithm (IRA)

```
1: for each demand pair $k \in K_u$ do
2:   for each link $e' \in E'$ do
3:     if $e'$ is ready for $k$ then
4:       Set the cost of link $e'$ to 0, $cost(e') = 0$
5:     if $e'$ is not ready for $k$ then
6:       Update the link cost $cost(e')$ according to ($\star$)
7:     if $\beta C_{e'} \leq B_k$ or $a(out(e')) > r_{out(e')}$ then
8:       Set the cost of link $e'$ to infinity, $cost(e') = \infty$
9:   Find shortest path between $s_k$ and $d_k$, if there are more than one
     shortest paths, randomly select one. Install the 4-tuple rules along the
     path. Update $a(v)$.
10:  Set $\beta C_{e'} = \beta C_{e'} - B_k$
```

link e', $cost(e')$ is shown below:

$$cost(e') = \begin{cases} w(out(e')) & \text{if } \theta^u_{out(e')} = 0 \ or \ \theta^u_{out(e')} > 1 \\ (t(out(e')) - 1)w(out(e')) & \text{if } \theta^u_{out(e')} = 1 \end{cases} \quad (\star)$$

For each newly activated link e', the corresponding OpenFlow rule has to be installed to the $out(e')$ to direct the traffic. If initially no other link of $out(e')$ is used, one OpenFlow entry $(s_u, *, d_u, n(e'))$ will be installed on $out(e')$, so $cost(e') = w(out(e'))$. However, if previously one egress link is activated on switch $out(e')$, that is, initially all the flows are forwarded to single output port. To activate a new link with a new output port, we require the all the flows that the switches carries to be fully specified so that they can be directed to the corresponding output ports. Hence $cost(e') = (t(v) - 1)w(out(e'))$. Finally, if previously more than one link is activated on switch $out(e')$, for each new activated egress link, a new entry $(s_k, i, d_k, n(e'))(k \in K_u)$ is installed to direct the flow.

An example is given in Figure 6(a): assume initially switch $s1$ carries two demand pairs $[00, 10]$ and $[01, 11]$ of u that have the same output port 4 ($\theta_v^u = 1$), therefore one entry is installed to route the flows as shown in Figure 6(b). Assume one more demand pair $[00, 10]$ is added and another egress link is used to direct this flow (output port is 5), then number of entries in the routing table increases by $t(v) - 1 = 3 - 1 = 2$. Therefore the cost to activate this new link is $2w(v)$, the new flow table is shown in Figure 6(c). Algorithm 1 reuses the links which are *ready* by setting the weights of these links to 0. The weights of other links are updated according to ($\star$). If the bandwidth consumption on e' exceeds the maximum limit $\beta C_{e'}$, the cost of e' is set to be infinity, $cost(e') = \infty$. Finally the path can be set up by finding the shortest path between the source and the destination hosts.

We now analyze the complexity of IRA. The for loop between line 3 to 8 in IRA determines the cost for each edge $e \in E$. In line 9, the shortest path is calculated between each s_k to d_k. Therefore, the overall complexity is

Algorithm 2 Detailed Search Algorithm (DSA)

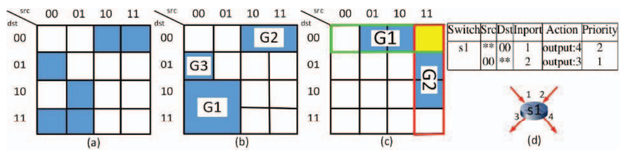
```
1: Set the source and destination address to the address with fully wildcard
   bit, set $u_{prev} = u_{curr} = \emptyset$, set $l_s = l_d = m$.
2: while $K \neq \emptyset$ do
3:     Set $prev = \infty$ and $curr = 0$
4:     while $curr \leq prev$ or $size(u_{curr}) > L$ do
5:         Set $[u_{s0}, avgcost(u_{s0})] = FindCost(src, l_s, 0)$
6:         Set $[u_{s1}, avgcost(u_{s1})] = FindCost(src, l_s, 1)$
7:         Set $[u_{d0}, avgcost(u_{d0})] = FindCost(dst, l_d, 0)$
8:         Set $[u_{d1}, avgcost(u_{d1})] = FindCost(dst, l_d, 1)$
9:         Select $u_{curr}$ equals to $u \in \{u_{s0}, u_{s1}, u_{d0}, u_{d1}\}$ with the min-
           imum $avgcost(u)$, if more than one such $u$ exist or all the
           $avgcost(u)$ equals infinity, randomly pick one.
10:        Set $curr = avgcost(u_{curr})$
11:        if $(curr > prev$ or $l_s = l_d = 0)$ then
12:            Remove all the demand pairs in $u_{prev}$ from $K$, building the
               path for each demand pair in $u_{curr}$ by using $IRA$.
13:            Set the source and destination address to full wildcard bits. Set
               $u_{prev} = u_{curr} = \emptyset$, $l_s = l_d = m$
14:            break
15:        Set the binary digit on leading bit according to $u_{curr}$, update the
           leading bit by decreasing $l_s$ or $l_d$ by 1 according to $u_{curr}$, set
           $prev = curr$, $u_{prev} = u_{curr}$
16: Function $FindCost$ (type, l, d)
17: if $(type == src$ and $l_s \neq 0)$ then
18:     Set the binary digit on leading bit $l$ of source address to $d$, while keeps
        destination address the same. Denote the routing group formed $u$.
19:     if $(0 < size(u) \leq L)$ then
20:         Reset the binary digit on the leading bit $l$ of the source address to
            wildcard bit.
21:         Return $[u, \frac{IRAcost(u)}{size(u)}]$
22:     if $(size(u) > L)$ then
23:         Return $[u, \infty]$
24: if $(type == dst$ and $l_d \neq 0)$ then
25:     Set the binary digit on the leading bit $l$ of the destination address to
        $d$, while keeps the source address the same. Denote the routing group
        formed $u$.
26:     if $(0 < size(u) \leq L)$ then
27:         Reset the binary digit on the leading bit $l$ of the destination address
            to wildcard bit.
28:         Return $[u, \frac{IRAcost(u)}{size(u)}]$
29:     if $(size(u) > L)$ then
30:         Return $[u, \infty]$
31: Return $[\emptyset, \infty]$
32: EndFunction
```

$\mathcal{O}(|K_u|(|V| + |E| \log |E|))$.

VI. Efficient Partitioning Problem

After solving ERP for each routing group, we are left with the problem of partitioning K demand pairs into routing groups. In this case all demand pairs can be visualized using $2^m \times 2^m$ square, where m is the number of bits in the source and destination address. For example, Suppose there are 6 demand pairs $[10, 00], [11, 00], [00, 01], [00, 11], [01, 11], [01, 10]$, the corresponding square is shown in Figure 7(a). The squares representing the 6 demand pairs are coloured in blue. One of the possible partitions is shown in Figure 7(b), where the routing group $G1$ covers the demand pairs $[01, 10], [01, 11], [00, 11]$, $G2$ covers $[10, 00], [11, 00]$ and $G3$ covers $[00, 01]$.

The goal of EPP is to find the routing groups so each group can be routed with the lowest cost as defined in (1). We represent each routing group by a pair of source-destination



Switch	Src	Dst	Inport	Action	Priority
s1	**	00	1	output:4	2
	00	**	2	output:3	1

Fig. 7. Examples

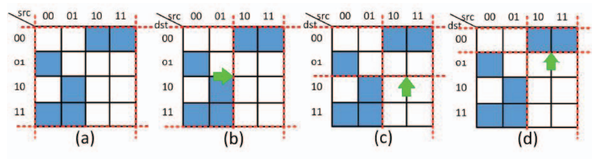


Fig. 8. Example of DSA

addresses with subnet mask. For example, $G2$ in Figure 7(b) can be represented by $[1*, 00]$. A drawback of aggregating flow entries is that we lose visibility into fine-grained flow characteristics, which makes elephant flow detection and rerouting harder to achieve [6]. In Sector, we use maximum routing group size L to limit the maximum flow aggregation level, which allows the Sector to make a trade-off between flow visibility and TCAM space savings.

Our solution algorithm, called *Detailed Search Algorithm (DSA)*, starts from the entire square that covers all source-destination pairs. In each iteration, it reduces the size of the routing rectangle by replacing the wildcard bit in the address with a binary digit. *The output of each iteration is the routing group with the lowest average cost per demand pair in the group.* Define the *leading bit* of an address as the leftmost wildcard bit in the address. For example, the leading bit of address $00**$ is the third bit. If there is no wildcard bit in the address, set the leading bit to 0. Denote $size(u)$ the number of demand pairs in routing group g. Define l_s and l_d as the leading bit of source and destination address. The pseudo code of *Detailed Search Algorithm* is described in algorithm 2. The function $IRAcost(u)$ returns the minimum cost generated by IRA to route all the demand pairs in u. The *DSA* algorithm works by searching the routing group u' with $size(u') < L$ with the lowest average cost in a greedy fashion, and building the paths for that group with minimum cost. Afterwards, the demand pair is removed from K. The algorithm terminates when all the demand pairs in K have been routed.

Figure 8 provides an example to illustrate DSA. Let L equal 3. Initially there are 6 demand pairs. The routing group is the region circled by the red dash line, which is the whole square shown in Figure 8(a). Assume we found the routing group with minimum average cost is $[1*, **]$, by setting the leading bit of source address to 1, the corresponding routing group is shown in Figure 8(b). Repeat these steps until we found the routing group $[1*, 00]$ shown in Figure 8(c) and 8(d). Note that further dividing this routing group will increase the average routing cost per demand pair. Then the two demand pairs in the routing group $[1*, 00]$ will be routed by using IRA. DSA then removes this routing group, and repeat the process until all the demand pairs are routed. We now analyze the complexity of DSA. The inner **while** loop between line $4-15$ runs at most $2m$ times, since in each iteration of the inner while loop the leading bit of source address or destination address decreases by 1, the iteration will stop when all the wildcard bit in source address and destination address are filled with binary digits. For each inner while loop, the IRA is called 4 times (line $5-8$). Finally, the outer wile loop (line $2-15$) runs at most $|K|$ times. Therefore the complexity of DSA is $\mathcal{O}(8m|K|^2(|V|+|E|\log|E|))$. It is possible that two routing groups may overlap with each other. For the example shown in Figure 7(c), two routing groups $G1$ and $G2$ both cover the yellow square $[11, 00]$. Assume the switch $s1$ carries the traffic of both routing groups, the flow of $[11, 00]$ will satisfy the predicates for both entries, which is shown in Figure 7(d). Therefore each entry in the switch must be assigned with a priority level. Upon receiving a packet, the switch finds the entries with a matching predicates and the highest priority level, then performs its action. One simple way to assign priorities in DSA is based on the order the routing group is generated by DSA. For example, if $G1$ is generated before $G2$, then the entry of $G1$ has a higher priority than that of $G2$ (shown in Figure 7(d)).

VII. Dynamic scheduling of demand pairs

The algorithms presented in the previous sections have been focused on the static version of the problem. While they are useful for networks that have constant network demand, in reality, the demand pairs may join/leave the network dynamically. In this section we propose the dynamic algorithms to deal with this scenario.

A. Dynamic demand pairs entering

We first consider the case where a new demand pair k enters the network. Let s_k and d_k denote the source and destination address of k. We first make the following definition:

Definition 2. Let f be a full address without wildcard bits, we say the address f' *covers* f if f' and f have the same bit length and all the non-wildcard bits of f' are the same as f.

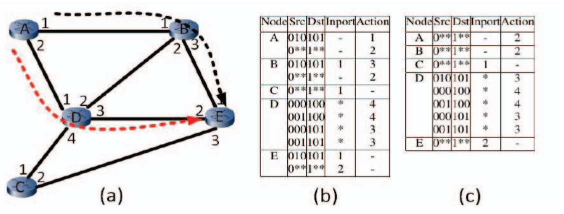
For example, let $f' = 00**$ and $f = 0001$, then f' covers f because all the non-wildcard bits of f' (the first two bits) are the same as f, which is 00. Next we extend the definition of *ready* for each new demand pair k:

Definition 3. In a directed graph $G' = (V', E')$, link e' is *ready* for the new demand pair k if: 1. $out(e')$ contains a 4-tuple $(s, i, d, n(e')), i \in \pi(out(e'))$ or $(s, *, d, n(e'))$. 2. $in(e')$ contains a 4-tuple $(s, n(e'), d, j), j \in \pi(in(e'))$ or $(s, *, d, j)$, where s *covers* s_k and d *covers* d_k.

Algorithm 3 ($DANA$) builds the paths for each new demand pair. The intuition behind $DANA$ is reusing existing rules in the network. For the example shown in Figure 1(c), the routing tables are shown in Figure 2(c). Assume that there exists a new demand pair with source address/destination address 010/101 and ingress/egress switches are A and E. Further assume that the every link has enough remaining capacity to carry the flow of this demand pair such that (8) is obeyed, every switch has the same weight and enough TCAM space. One of the possible solutions is routing through the path

Algorithm 3 Dynamic Algorithm for New Arrivals (DANA)

```
1: for each new demand pair $k$ do
2:     for each link $e' \in E'$ do
3:         if $e'$ is ready for $k$ then
4:             Set the cost of link $e'$ to 0, $cost(e') = 0$
5:         if $e'$ is not ready for $k$ then
6:             Set the link cost $cost(e') = w(out(e'))$
7:         if $\beta C_{e'} \leq B_k$ or $a(out(e')) > r_{out(e')}$ then
8:             Set the cost of link $e'$ to infinity, $cost(e') = \infty$
9:     Find shortest path between $s_k$ and $d_k$, if there are more than one short-
       est paths, randomly select one. Install the 4-tuple rules $(s_k, i, d_k, j)$
       along the path. Update $a(v)$.
10:    Set $\beta C_{e'} = \beta C_{e'} - B_k$
```



(b)

Node	Src	Dst	Inport	Action
A	010	101	-	1
	0**	1**	-	2
B	010	101	1	3
	0**	1**	-	2
C	0**	1**	1	-
D	000	100	*	4
	001	100	*	4
	000	101	*	3
	001	101	*	3
E	010	101	1	-
	0**	1**	2	-

(c)

Node	Src	Dst	Inport	Action
A	0**	1**	-	2
B	0**	1**	-	2
C	0**	1**	1	-
D	010	101	*	3
	000	100	*	4
	001	100	*	4
	000	101	*	3
	001	101	*	3
E	0**	1**	2	-

Fig. 9. Example of Flow Tables

A, B, E (the black route in Figure 9(a)), and the new routing table is shown in Figure 9(b). Three entries are added on the switch A,B and E. $DANA$ will generate the red route shown in Figure 9(a) and the routing table shown in Figure 9(c). By comparison, only one entry is installed on switch D, and the entries in switch A, D are reused so that no additional entry is installed.

B. Dynamic demand pairs leaving

In case of a demand pair leaving the network, if the leaving renders the rule to be obsolete, this rule can be safely deleted either by the controller or idle timeout [10]. However, depending on the network traffic pattern, some unused rules can be kept for a longer time for routing future traffic flows. Details of this problem is out of the scope of the paper [17].

VIII. Simulations

A. Network Settings

We evaluated DSA on 4 different network topologies, one is a real WAN model generated by *GT-ITM* [18], which simulates WANs using Transit-Stub topologies. This network has 100 nodes and 127 undirected links. The other network topologies includes the *Abilene* (11 nodes, 13 undirected links), *Fat Tree* (4 pods, 4 core switch, 52 nodes and 64 undirected links) and *Sprint* (52 nodes, 168 undirected links). The traffic distribution for Abilene and Sprint are available in [19]. We use two models proposed in [19]: *Lognormal* distribution ($\mu = 15.45, \delta = 0.885$), and *Weibull* distribution ($a = 1.87 \times 10^5, b = 0.69$) to model the traffic distribution in the Sprint Network. And we use the *Lognormal* distribution ($\mu = 16.6, \delta = 1.04$) to model the traffic distribution in the Abilene Network. For the *GT-ITM* and *Fat Tree*, we use the Bimodal distribution (generated by mixture of two Gaussian Distributions) proposed in [20].

The Bimodal distribution is proposed based on the observation that only a small fraction of Source-Destination pairs has large flows. Assume each switch has a capacity between $300 - 500$ entries. We use the method proposed in [21] to model the link capacity, which claims that the link capacity distribution follows the *Zipf's Law*, and the links whose end nodes with higher degree tend to have larger link capacity. For the purpose of simulation, we set the link capacity to $39813.12Mbps$ (the transmission rate of optical carrier $OC768$) if the degrees of both endpoints of that link are larger than 3, set the link capacity to $9953.28Mbps$ ($OC192$) if one endpoint has degree larger than 3 and degree of the other end point is less or equal 3, set the link capacity to $2488.32Mbps$ ($OC48$) if the degree of both endpoints is less or equal than 3.

We randomly generate demand pairs, each corresponds to a source machine and destination machine in the network. Each machine has been assigned a random type B IP address and they are connected to a switch in the network. The bandwidth consumption of the flows follows the distribution described above. Since TCAM space aware routing has not been investigated before, and there is no such a similar routing algorithm which aims to reduce the routing table size, we compare *DSA* with two benchmark routing schemes: *ECMP* and *Valiant Load Balancing (VLB)* which are widely used to achieve load balancing on link resources. *ECMP* is a routing strategy which works by equally splitting the traffic over the multiple paths with the same length (number of hops) [22]. In *VLB*, the flows of the same demand pair are first sent to some intermediate nodes, then forwarded to the destination [23]. After the paths are calculated by the two benchmark routing schemes, the corresponding rules (s_k, i, d_k, j) are installed to direct the flows. All the rules contain fully specified addresses s_k and d_k so that they can not be reused by the other flows. We run each algorithm 100 times and take the average results. For the evaluation, we set the weight of each switch in (1) to 1, therefore the total cost generated by (1) equals the total number of entries installed. We compare performance of the algorithm using a metric called Traffic Saving Ratio. Assume the total amount of TCAM space consumed by *DSA* is T_p, and total amount of TCAM space consumed by the benchmark algorithm is T_b , then *Traffic Saving Ratio (TSP)* is defined as:

$$TSP = (T_b - T_p)/T_b \tag{9}$$

B. Evaluation of the TSP

First we evaluate the relation between the number of demand pairs and TSP. We do not limit the maximum routing group size. Table II shows the relations between the number of flows and TSP with different networks and different traffic distributions. TSP_1 is the TSP of the *ECMP* and TSP_2 is the TSP of the *VLB*. The *DSA* can achieve $20\% - 80\%$ saving on the TCAM space with different network topologies and traffic distributions. The saving also grows with the number of flows. This is because as the number of flows increases, more flows can be aggregated for saving TCAM space. Moreover, if we neglect the first packet of each flow which is forwarded to the controller, the TCAM space saving almost equals to the saving in the number of control traffic between the controller

TABLE II. Evaluate on All the Network Topologies

Network	Number of Flow		60	70	80	90	100
Abilene	Lognormal	TSP_1	0.4264	0.6331	0.7176	0.7518	0.7851
		TSP_2	0.4051	0.6318	0.7015	0.7427	0.7881
Network	Number of Flow		80	120	160	200	240
Sprint	Weibull	TSP_1	0.2570	0.3740	0.5252	0.5805	0.6659
		TSP_2	0.2551	0.4553	0.5038	0.6519	0.7718
	Lognormal	TSP_1	0.2041	0.2869	0.5437	0.5809	0.6003
		TSP_2	0.2473	0.3933	0.5154	0.6715	0.7258
Network	Number of Flow		200	300	400	500	600
GT-ITM	Bimodal	TSP_1	0.4253	0.4353	0.5766	0.6107	0.6611
		TSP_2	0.4542	0.4604	0.5599	0.6034	0.7912
Network	Number of Flow		100	150	200	250	300
Fat tree	Bimodal	TSP_1	0.1981	0.2986	0.4334	0.6008	0.6745
		TSP_2	0.2158	0.2974	0.4298	0.6177	0.7208

TABLE III. Performance and Running Time Comparison

Performance	Node	Abilene	Sprint	Tree	GT-ITM
	TSP	0.5979	0.5435	0.4894	0.4001
Running Time	Network	Abilene	Sprint	Tree	GT-ITM
	DSA	0.19ms	0.29ms	0.32ms	0.37ms

and the OpenFlow switches. This is because each entry in the switches requires a control packet for installation. Figure 10 to 13 show the relations between the number of flows and the actual maximum link utilization of different algorithms over different traffic distributions. When calculating the link utilization of *DSA*, the threshold on link utilization rate, β is set to 0.9. The maximum link utilization rate of *DSA* is on average $10 - 17$ percent higher than that of *ECMP* and *VLB*. Despite the higher link utilization rate of *DSA*, considering the huge savings on the TCAM space, we believe this is a fair trade-off.

As mentioned before, we can tune the value of L to balance the trade-off between TCAM space saving and maximum link utilization. We evaluate on the *GT-ITM* network with 500 demand pairs. As shown in Figure 14 and 15, when L decreases from 95 to 5, the TSP decreases from 0.6107 to 0.04665, meaning less demand pairs are aggregated. At the same time, the maximum link utilization rate also decreases slightly from 0.79 to 0.752. This is because small routing group leads to fine-grained routes which in turn reduces maximum link utilization.

C. Evaluation of the DANA

We generate some demand pairs which attach to some random nodes in the network. Each demand pair has a random type B source and destination IP address, and all the demand pairs are connected by installing the rules generated by DSA. To emulate the dynamic entering of new demand pairs, we generate 50 new demand pairs and run the DANA to add the flows. We compare the performance of DANA with *shortest path algorithm (SPA)*, which routes traffic along shortest paths. All the rules installed for SPA are fully specified addresses. TSP is defined in a similar manner as (9), with T_b and T_p means the total number of rules generated by SPA and DANA to direct the new flows. Table III shows the performance and running time of the two algorithms. As the tables shows, DANA can achieve $40\% - 60\%$ saving on TCAM space. The running time of DANA increases moderately with the

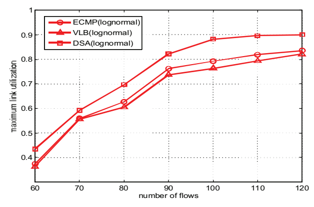


Fig. 10. Link utilization (Abilene)

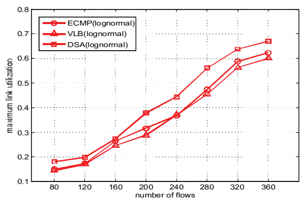


Fig. 11. Link utilization (Sprint)

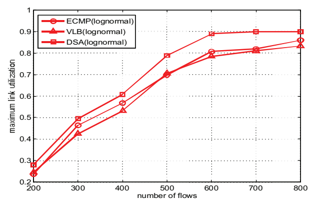


Fig. 12. Link utilization (GT-ITM)

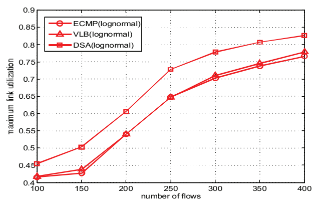


Fig. 13. Link utilization (Fat Tree)

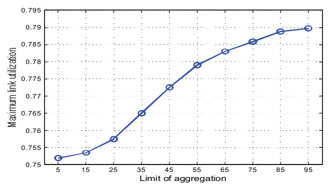


Fig. 14. Change on link utilization

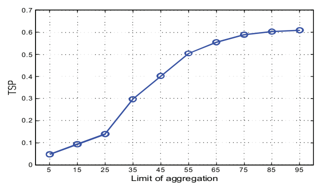


Fig. 15. TSP on GT-ITM

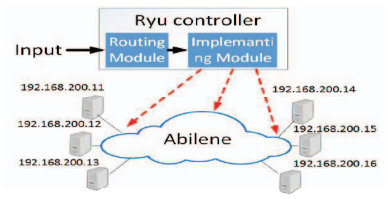


Fig. 16. Real Testbed Experiment

network size. But overall running time of the algorithm is still reasonable.

IX. Testbed Deployment

We evaluated the functionality and implementability of the DSA in a real testbed. We built the overlay network that follows Abilene network topology by using software switch (OpenVswitch) running on virtual machines. The OpenVswitches communicate with each other by using the Virtual Extensible LAN. The centralized controller (Ryu) can configure the entries in the switches to build the routing paths. We built three source VMs and three destination VMs (three demand pairs), each VM is assigned with an IP address. The routing module on top of the Ryu controller takes the connection demands as the input and sends the results of DSA to the implementing module which installs the relative OpenFlow rules on the switches (Figure 16). For comparison, we also used the shortest path algorithm (Dijkstra's Algorithm) to connect the demands pairs. For DSA, total 14 entries are installed on the switches, and the total time taken for building the path is 0.028s. For Dijkstra's Algorithm, total 30 entries are installed on the switches with the total time 0.061s. Hence, DSA clearly saves TCAM space and path set up time.

X. Conclusions

In this paper, we propose *Sector*, a routing scheme to achieve savings on TCAM space in SDN without causing network congestions. We provide algorithms for this problem for both the static and dynamic scenarios. Experiments show that *Sector* can achieve $20\% - 80\%$ saving on TCAM space with $10\% - 17\%$ increase in maximum link utilization.

References

[1] N. Kang, Z. Liu, J. Rexford, D. Walker. *"Optimizing the 'One Big Switch' Abstraction in Software-Defined Networks"*, in ACM Conext, 2013.

[2] C.Y Hong, S.Kandula, R.Mahajan, et al. *"Achieving High Utilization with Software-Driven WAN"*, in Proceedings of ACM Sigcomm, 2013

[3] M. Zhang, et al.*"GreenTE: Power-Aware Traffic Engineering"*, IEEE ICNP, 2013.

[4] E. Oki, et al.*"Fine Two-Phase Routing with Traffic Matrix"*, in IEEE ICCCN, 2009.

[5] X. Liu, R. Meiners,E. Torng. *"TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs"*. IEEE/ACM transaction on networking, VOL. 18, NO. 2, APRIL 2010.

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, et al. *"DevoFlow: Scaling flow management for high-performance networks*, in ACM Sigcomm, 2011.

[7] P. Lekkas. *"Network Processors : Architectures, Protocols and Platforms"*, Mc-Graw Hill Professional, Jul 28, 2003

[8] Y. Kanizo, D. Hay, I. Keslassy. *"Palette: Distributing tables in software-defined networks"*, in Proceedings of IEEE Infocom, 2013.

[9] M. Alizadeh, A. Greenberg, et al. *DCTCP:Efficient packet transport for the commoditized data center.* In ACM SIGCOMM, 2010.

[10] Openflow Spec. www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[11] M. Moshref, M. Yu, A. Sharma, R. Govindan. *"Scalable Rule Management for Data Centers"*, in Proceedings of USENIX NSDI, 2013.

[12] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. *"Scalable Flow-Based Networking with DIFANE"*. In Proceedings of Sigcomm, 2010.

[13] S.Yeganeh, Y.Ganjali. *"Kandoo: a framework for efficient and scalable offloading of control applications"*, in ACM HotSDN, 2012.

[14] A.S Tam, et al. *"Use of devolved controllers in data center networks"*, in Infocom Computer Communications Workshops, 2011.

[15] K. Phemius, M. Bouet and J. Leguay. *"DISCO: Distributed multi-domain SDN controllers"*, in Proceedings of IEEE NOMS, 2014.

[16] DP. Williamson, DB. Shmoys. "The Design of Approximation Algorithms": http://www.designofapproxalgs.com/

[17] H. Zhu, H. Fan, X. Luo, Y. Jin. *"Intelligent Timeout Master: Dynamic Timeout for SDN-based Data Centers"*, in IEEE IM, 2015.

[18] GT-ITM website: www.cc.gatech.edu/projects/gtitm/

[19] A. Nucci, et al. *"The problem of synthetically generating IP traffic matrices: initial recommendations"*, ACM Sigcomm CCR, July 2005.

[20] A. Medina, N. Taft, et al. *"Traffic matrix estimation: existing techniques and new directions"*, in the Proceedings of SIGCOMM, 2002.

[21] T. Hirayama, S. Arakawa, S. Hosoki, and M. Murata, *"Models of link capacity distribution in ISPs router-level topologies"*, JCNC, 2011.

[22] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992.

[23] R. Zhang-Shen. *"Valiant Load-Balancing: Building Networks That Can Support All Traffic Matrices"*, Springer, 2010.

[24] R. Cohen, L. Lewin-Eytan, J. Naor, D. Raz, *"On the effect of forwarding table size on SDN network utilization"*, in Infocom 2014.

[25] N. Katta, et al. *"Infinite CacheFlow in software-defined networks "*, in the Proceedings of HotSDN, 2014.