Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking

Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen
Department of Computer Science and Information Engineering
National Central University
Jhongli City, Taiwan

***Abstract*—This paper extends the well-known Dijkstra's shortest path algorithm to consider not only the edge weights but also the node weights for a graph derived from the underlying SDN topology. We use Pyretic to implement the extended Dijkstra's algorithm and compare it with the original Dijkstra's algorithm and the unit-weighted Dijkstra's algorithm under the Abilene network topology in terms of end-to-end latency with the Mininet tool. As shown by the comparisons, the extended Dijkstra's algorithm outperforms the other algorithms.**

Keywords—software defined networking (SDN); shortest path; Dijkstra's algorithm; network topology

I. Introduction

Software Defined Networking (SDN) is a concept to decouple the control plane and data plane of network devices [8][14]. Mckeown et al. proposed the OpenFlow protocol to realize the SDN concept to allow researchers to experiment novel network protocols [6]. In SDN, a logically centralized controller configures the forwarding tables (also called flow tables) of switches, which are responsible for forwarding the packets of communication flows. In this way, SDN users can composite application programs run on top of the controller to monitor and manage the whole network in a centralized and real-time manner.

The emergence of the SDN technology brings many new network applications realized by programming the SDN controller. Typical examples include load balancing, multimedia multicast, intrusion detection, and so on. It is also usefull for realizing network virtualization [2]. Some researchers developed programming languages, such as Frenetic [3] and Pyretic [11], to facilitate SDN application design. Frenetic is a declarative query language for classifying network traffic and providing a functional reactive combinator library for describing high-level packet-forwarding policies [3]. Pyretic is a Python-base language that is extended from Frenetic. Pyretic raises the level of network abstraction and enables programmers to create modular software for SDN [3].

In this paper, we extend the well-known Dijkstra's shortest path algorithm [1] to consider not only the edge weights but also the node weights for a graph derived from the underlying SDN topology. We use Pyretic to implement the extended Dijkstra's algorithm and compare it with the original Dijkstra's algorithm and the unit-weighted Dijkstra's algorithm under the Abilene network [12] in terms of end-to-end latency with the Mininet tool. As shown by the comparisions, the extended Dijkstra's algorithm outperforms the other algorithms.

We have noticed that the study [4] has addressed the implementation issues for the modified Dijkstra's algorithm [10] and the modified Floyd-Warshall shortest path algorithm in OpenFlow. However, the modified Dijkstra's algorithm is different from the proposed extended Dijkstra's algorithm in the sense that the former is modified to solve the multi-source single-desitination shortest path problem and the latter is extended from the Dijkstra's algorithm to consider both edge weights and node weights for solving the single-source shortest path problem. Hence, we will not compare the proposed algorithm with the modified Dijkstra's algorithm. It is worth mentioning that the extension concept proposed in this paper can also be applied to the modified Dijkstra's algorithm.

The remainder of this paper is organized as follows. In Section II, we introduce some preliminary knowledge, including the SDN concept, Pyretic, and Mininet. Section III describes the extended Dijkstra's algorithm and its implementation. Section IV shows the simulation results and observations. Finally, this paper is concluded with Section V.

II. Preliminaries

A. Software Defined Networking

SDN advocates the separation of control and data planes (or layers), where underlying switching hardware devices (called *switches*) are controlled via software entities (called *applications*) that runs in external, decoupled automated control plane devices (called *controllers*) [14]. Fig. 1 depicts the logical view of the SDN architecture. SDN enables network administrator to write applications to manage network services, including routing, access control, multicast, and other traffic engineering tasks.

OpenFlow is one of the first open protocols defined between the control plane device, the controller, and the data plane device, the *switch*, of the SDN architecture [14]. An OpenFlow switch consists of one or more *flow tables* and/or *group tables*, as shown in Fig. 2. An OpenFlow controller can update, add and delete flow entries in flow table both reactively and proactively. Each flow table in the switch contains a set of flow entries, each of which consists of *match fields, counters,* and *set of instructions*, as shown in Fig. 3.

On receiving a packet, a switch first matches it with entries in the flow table(s). The matching process begins in the first table and continues subsequently to the additional tables. It is prioritized; that is, the first matched entry in each table is to be returned. If a matched entry is found, the instructions associated with the entry are executed to complete actions, such as forwarding the packet to another switch via a certain port, matching the packet in another table, dropping the packet, etc. If no match is found in any flow table, the outcome depends on the configuration of the table, and the packet may be forwarded to the controller over the OpenFlow channel, be dropped, or be sent to the next flow table for matching [9].

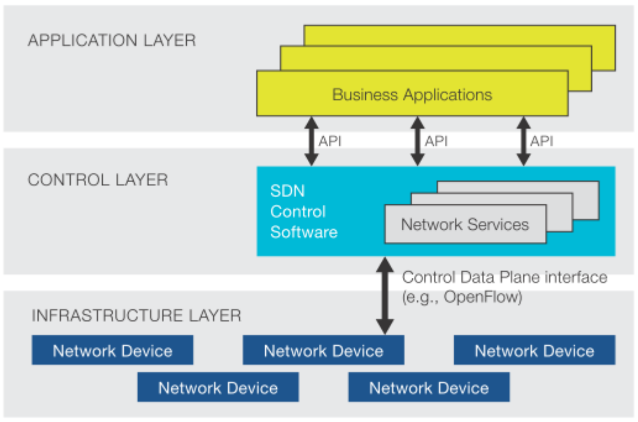

Fig. 1. The illustration of the SDN architecture [14]

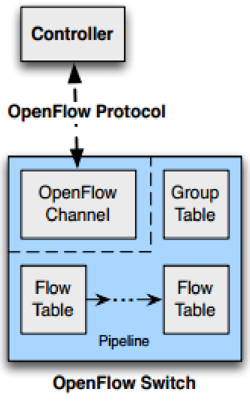

Fig. 2. The OpenFlow controller and the switch [9]

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Fig. 3. The flow table entry of the OpenFlow switch [9]

B. *Pyretic*

Frenetic is introduced as a high-level language to program the controller to manage switches in the SDN network [3]. Frenetic provides programmers with a declarative query interface for classifying and aggregating network traffic as well as a functional reactive combinator library for describing high-level packet-forwarding policies.

Based on Frenetic and Python, Pyretic is introduced as an SDN programming language or platform that raises the level of abstraction [11][7]. It allows programmers to focus on how to specify a network policy at high-level abstraction. For example, Pyretic hides low-level details by allowing programmers to express policies as compact, abstract functions that take a packet as input, and return a set of new packets. Pyretic also facilitates modular design by offering two policy composition operators, *parallel composition* and *sequential composition*, to allow programmers to combine multiple policies together without worrying about their conflicts. Pyretic programmers can also create a dynamic policy whose behavior will change over time. To sum up, Pyretic enables SDN programmers to create succinct modular network applications at a high-level of abstraction.

C. *Mininet*

Mininet [5][13] is an open source network emulator that supports the OpenFlow protocol for the SDN architecture. It is one of the most popular tools used by the SDN research community. It uses the virtualization approach to create a network of virtual hosts, switches, controllers, and links. a realistic virtual network. Jus as an operating system virtualizes computing resources with process abstraction, Mininet uses process-based virtualization to emulate entities on a single OS kernel by running real code, including standard network applications, the real OS kernel and the network stack. Therefore, a design that works properly in Mininet can usually move directly to pratical networks composed of real hardware devices.

Mininet provides a ready way to get the SDN network behavior and performance for different experimental network topologies. It enables complex topology testing without wiring up a physical network. It supports not only a basic set of parametrized topologies, but also arbitrary custom topologies. We can even create custom topologies by writing Python scripts in Mininet, since an extensible Python API for network creation and experimentation is offered by Mininet, whose code is almost entirely in Python.

III. The Extended Dijkstr's Algorithm Implementation

A. *Description of the Extended Dijkstra's Algorithm*

Given a weighted, directed graph $G=(V, E)$ and a single source node s, the classical Dijkstra's algorithm can return a shortest path from the source node s to every other node, where V is the set of nodes and E is the set of edges, each of which is associated with a non-negative *weight* (or *length*). In the original Dijkstra's algorithm, nodes are associated with no weight. Below we show how to extend the original algorithm to consider both the edge weights and the node weights.

Fig. 4 shows the extended Dijkstra's algorithm, whose input is a given graph $G=(V, E)$, the edge weight setting ew, the node weight setting nw, and the single source node s. The extended algorithm uses $d[u]$ to store the *distance* of the current shortest path from the source node s to the destination node u, and uses $p[u]$ to store the *previous* node preceding u on the current shortest path. Initially, $d[s]=0$, $d[u]=\infty$ for $u\in V$, $u\neq s$, and $p[u]=$null for $u\in V$.

Note that the extended Dijkstra's algorithm is similar to the original Dijkstra's algorithm. The difference is that we add the node weight in line 6 and line 7 of the algorithm. In light of Dijkstra's work, we can prove that the extended algorithm indeed return the shortest path from the single source node to every other node with the consideration of the edge weights and node weights. To save space, we omit the proof of the above statement.

Also note that the original Dijkstra's algorithm cannot achieve the same result just by adding node weights into edge weights. This is because the node weight should be considered only at the outgoing edge of an intermediate node on the path. Adding node weights into edge weights implies that an extra node weight of the destination node is added into the total weight of every shortest path, making the algorithm return the wrong result.

```
Extended Dijkstra's Algorithm
Input: $G=(V, E)$, $ew$, $nw$, $s$
Output: $d[|V|]$, $p[|V|]$
1:  $d[s]\leftarrow 0$; $d[u]\leftarrow\infty$, for each $u\neq s$, $u\in V$
2:  insert $u$ with key $d[u]$ into the priority queue $Q$, for each $u\in V$
3:  while ($Q\neq\emptyset$)
4:    $u\leftarrow$Extract-Min($Q$)
5:    for each $v$ adjacent to $u$
6:      if $d[v] > d[u]+ew[u,v]+nw[u]$ then
7:        $d[v]\leftarrow d[u]+ew[u,v]+nw[u]$
8:        $p[v]\leftarrow d[u]$
```

Fig. 4. The extended Dijkstra's algorithm

B. The Application of the Eextended Dijkstra's Algorithm

The extended Dijkstra's algorithm is very useful in deriving the best routing path to send a packet from a specific source node to another node (i.e., the destination node) for the SDN environment in which significant latency occurs when the packet goes through intermediate nodes and edges (or links). Below, we show how to define the edge weights and node weights so that the extended Dijkstra's algorithm can be applied to derive routing path for some specific SDN environment.

Assume that we can derive from the SDN topology a graph $G=(V, E)$, which is weighted, directed, and connected. For a node $v\in V$ and an edge $e\in E$, let $Flow(v)$ and $Flow(e)$ denote the set of all the flows passing through v and e, respectively, let $Capacity(v)$ be the *capacity* of v (i.e., the number of bits that v can process per second), and let $Bandwidth(e)$ be the bandwidth of e (i.e., the number of bits that e can transmit per second). The node weight $nw[v]$ of v is defined according to Eq. (1), and the edge weight $ew[e]$ of e is defined according to Eq. (2).

$$nw[v] = \frac{\sum_{f\in Flow(v)} Bits(f)}{Capacity(v)}, \qquad (1)$$

where $Bits(f)$ stands for the number of f's bits processed by node v per second.

$$ew[e] = \frac{\sum_{f\in Flow(e)} Bits(f)}{Bandwidth(e)}, \qquad (2)$$

where $Bits(f)$ stands for the number of f's bits passing through edge e per second.

Note that we can easily obtain the number of a flow's bits processed by a node or passing through an edge with the help of the "counters field" of the OpenFlow switches' flow tables. Also note that the numerators in Eq. (1) and Eq. (2) are of the unit of "bits", and the denominators are of the unit of "bits per second". Therefore, the node weight $nw[v]$ and the edge weight $ew[e]$ are of the unit of "seconds". When we accumulate all the node weights and all the edge weights along a path, we can obtain the end-to-end latency from one end to the other end of the path.

IV. Simulation

A. Simulation Setting

We use Mininet [5][13] and adopt the topology of the Abilene network [12] to perform simulation. The Abilene network is a high-performance backbone network suggested by the Internet2 project. Fig. 5 shows a historical Abilene (network) core topology [15], connecting 11 regional sites or nodes across the United States. The Abilene network has 10 Gbps connectivity between neighboring nodes and 100 Mbps connectivity between a host and a node.

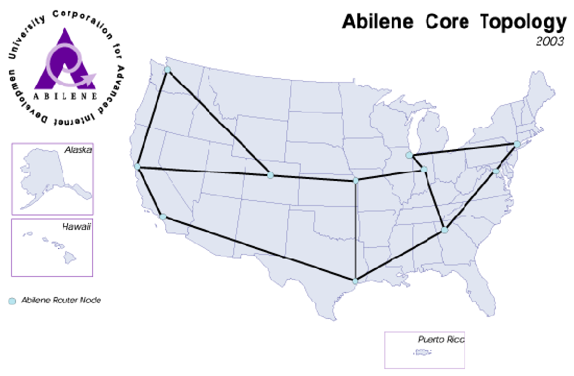

Fig. 5. The Abilene network core topology [15]

Based on the Abilene core topology, we set up in Mininnet an SDN/OpenFlow network with one controller and 11 switches as nodes, where each switch is linked to the controller logically and is attached with one host, as shown in Fig. 6. The simulation parameter settings are shown in TABLE I.

We use Iperf as a testing tool to generate TCP data streams in our simulation. The testing time for every testing case is 1000 seconds. In the Abilene topology generated in Mininet, as shown in Fig. 6, the host 1 with a red circle is set as an Iperf server, and all other 10 hosts are set as Iperf clients. When the simulation starts, all clients use the Iperf client command to repeatedly send packets to host 1, the Iperf server, at the same time, while host 1 uses the Iperf server command to receive packets from all clients for measuring performance. The Iperf tool reports the average TCP bandwidth between a client and the server, and we use the packet size of 53 bytes to derive the average end-to-end latency. There are 10 Iperf clients in our simulation; however, we focus on measuring the latency of one representative host, namely, the host 10 with a green circle, as shown in Fig. 6.

B. Simulation Resutls

When the Iperf testing runs, three shortest path algorithms, namely, the original Dijkstra's algorithm, the extended Dijkstra's algorithm, and the unit-weighted Dijkstra's algorithm, are separately used to generate both a client-to-server path and a server-to-client path for every client. By the unit-weighted Dijkstra's algorithm, we mean the original Dijkstra's algorithm taking all edge weights as 1. Hence, the unit-weighted Dijkstra's algorithm will return the path with the minimum hop counts for a pair of a source node and a destination node.

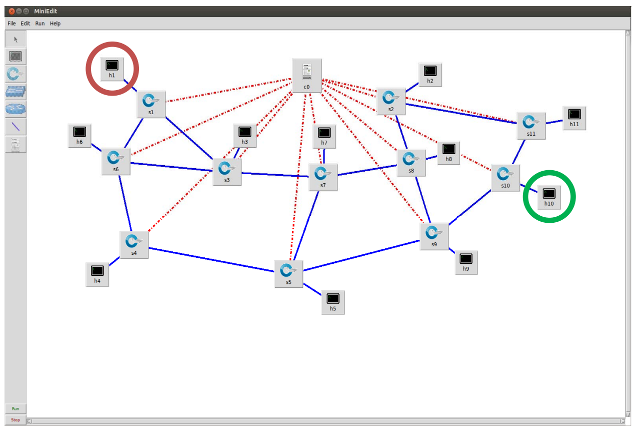

Fig. 6. Setting up the Abilene topology in Mininet

TABLE I. SIMULATION SETTINGS

Parameter	Setting
Bandwidth on edges	100Mbps ~ 1Gbps
Capacity of nodes	3Gbps ~ 7Gbps
Number of hosts	10
Number of nodes	11
Number of edges	25
Controller	POX 2.0 supporting Pyretic
Testing tool	Iperf
Testing time per case	1000 sec

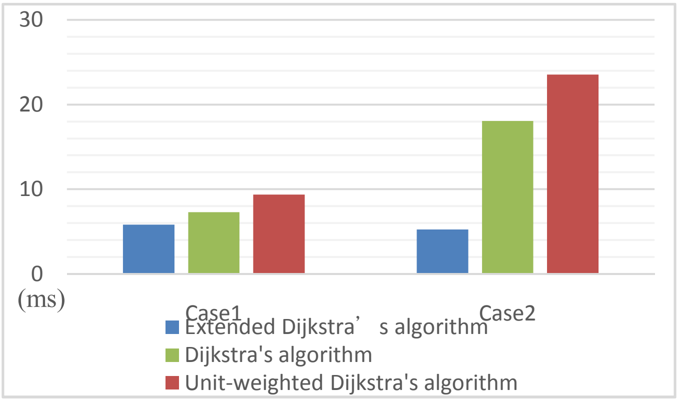

Fig. 7. The end-to-end latency of an Iperf server and a client

We have two simulation cases. For either case, the bandwidth of an edge and the capacity of a node are set randomly to be within the range shown in TABLE I. The simulation results are shown in Fig. 7. By the simulation results, we can see that the Extended Dijkstra's algorithm has smaller end-to-end latency than the original Dijkstra's algorithm, which in turn has smaller end-to-end latency than the unit-weighted Dijkstra's algorithm does. This is because the extended Dijkstra's algorithm considers both the edge weights and node weights, and the original Dijkstra's algorithm only considers the edge weights, and the unit-weighted Dijkstra's only considers the number of hops to generate the shortest path between a pair of hosts.

V. CONCLUSION

In this paper, we have extended the well-known Dijkstra's shortest path algorithm to consider both edge weights and node weights for a graph derived from the underlying SDN topology. We have implemented the extended Dijkstra's algorithm in Pyretic and compared it with the original Dijkstra's algorithm and the unit-weighted Dijkstra's algorithm under the Abilene network topology in terms of end-to-end latency with the Mininet tool. As shown by the comparisions, the extended Dijkstra's algorithm outperforms the other algorithms. In the future, we plan to conduct more comprehensive simulation experiments for more simulation cases under more SDN topologies by using more simulation tools to show the advantages of the extended Dijkstra's algorithm.

REFERENCES

[1] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no.1, 1959, pp. 269-271.

[2] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, 2013, pp. 20-27.

[3] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language", in *Proc. of the 16th ACM SIGPLAN International Conference on Functional Programming*, 2011, pp 279-291.

[4] A. Furculita, M. Ulinic, A. Rus, and V. Dobrota, "Implementation issues for Modified Dijkstra's and Floyd-Warshall algorithms in OpenFlow," in *Proc. of 2013 RoEduNet International Conference 12th Edition: Networking in Education and Research*, 2013, pp. 141-146

[5] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proc. of ACM Hotnets'10*, 2010.

[6] N. McKeown, et. al., "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication*, 2008.

[7] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," in *Proc. of NSDI*, 2013.

[8] B. Nunes, M. Mendonça, X. Nguyen, K. Obraczk, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," to appear in *IEEE Communications Surveys & Tutorials*, 2014.

[9] Open Networking Foundation, "OpenFlow Switch Specification version 1.4.0," October 14, 2013.

[10] A. Rus, V. Dobrota, A. Vedinas, G. Boanea, and M. Barabas, "Modified Dijkstra's algorithm with cross-layer QoS," *ACTA TECHNICA NAPOCENSIS, Electronics and Telecommunications*, vol. 51, no. 3, 2010, pp. 75-80.

[11] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic", *Technical Reprot of USENIX*, available at http://*www.usenix.org*, 2013.

[12] Abilene Network, http://en.wikipedia.org/wiki/Abilene_Network-#cite_note-line-1, last accessed on March 4, 2014.

[13] Mininet Website, http://mininet.org/, last accessed on May 2014.

[14] Open Network Foundation (ONF) Website (SDN white paper), https://www.opennetworking.org/sdn-resources/sdn-definition, last accessed on January 2014.

[15] Historical Abilene Connection Traffic Statistics, http://stryper.uits.iu.edu/abilene/, last accessed in March 2014.

[16] Iperf, http://iperf.fr/, last accessed in May 2014.