Fast Shortest-Path Queries on Large-Scale Graphs

Qiongwen Xu*†, Xu Zhang*†, Jin Zhao*†, Xin Wang*† and Tilman Wolf‡
*School of Computer Science, Fudan University, China
†Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China
‡Department of Electrical and Computer Engineering, University of Massachusetts Amherst, USA
{qwxu15, xuzhang09, jzhao, xinw}@fudan.edu.cn, wolf@umass.edu

Abstract—**Shortest-path queries on weighted graphs are an essential operation in computer networks. The performance of such algorithms has become a critical challenge in emerging software-defined networks (SDN), since SDN controllers need to perform a shortest-path query for every flow. Unlike classic solutions (e.g., Dijkstra's algorithm), high-performance shortest-path query algorithms include two stages: preprocessing and query answering. Despite the improved query answering time, existing two-stage algorithms are still extremely time-consuming in preprocessing large-scale graphs. In this paper, we propose an efficient shortest-path query algorithm, called BBQ, which reduces the running time of both stages via tree decomposition. BBQ constructs a distance oracle in a bottom-top-bottom manner, which significantly reduces preprocessing time over existing algorithms. In addition, BBQ can answer batch queries in bulk by traversing the decomposed tree instead of executing separate queries. Our experimental results show that BBQ outperforms state-of-the-art approaches by orders of magnitude for the running time of the preprocessing stage. Meanwhile, BBQ also offers remarkable acceleration for answering batches of queries. As a result, SDN controllers that use BBQ can sustain 1.1–27.9 times higher connection request rates.**

I. Introduction

Finding the shortest path in a weighted graph is a fundamental problem in networking (e.g., network traffic engineering [1]) and in many other domains (e.g., navigation [2], robotic control systems [3]). A number of well-known algorithms, such as Dijkstra, Bellman-Ford, and Floyd-Warshall are used in practice to provide solutions to this problem. However, with the emergence of *software-defined networks (SDN)*, traditional algorithms have reached performance limitations.

In SDN, the control plane functionality of the network is concentrated in the SDN controller, which in turn instructs SDN switches how to forward traffic in the data plane. The SDN controller maintains the network state, including the connection topology of switches and the transmission costs of links. For every connection that traverses the SDN, the controller needs to compute a path through the SDN [4], which translates into a shortest-path query on a weighted graph. In large-scale SDN, controllers need to consider topologies with thousands of switches [5] and accommodate thousands of connection requests per second. The shortest-path computation on the network graph is the computationally most demanding step in this process and thus presents a critical performance bottleneck for SDN deployment.

Classic solutions to the shortest-path query problem are based on *single-source shortest path (SSSP)* or *all-pairs shortest paths (APSP)* algorithms. SSSP algorithms take $O(n^2)$ time to answer a single query (e.g., Dijkstra's algorithm[1]) [6], where n is the number of nodes in the graph. APSP algorithms (e.g., Floyd-Warshall's algorithm) take constant time to answer a single query, but spend $O(n^3)$ time to calculate all possible shortest-path queries, and also require $O(n^2)$ space to store a pre-computed table of the shortest paths between all pairs of vertices [7]. The slow query time for SSSP algorithms and the expensive space consumption for APSP algorithms are not suitable for fast shortest-path queries on large-scale graphs.

Recently, more efficient shortest-path query algorithms have been developed [8]–[10]. Different from conventional SSSP and APSP algorithms, these algorithms have two stages: *preprocessing* and *query answering*. At the preprocessing stage, a *distance oracle* is pre-computed, which allows for fast retrieval of the shortest path for any pair of vertices. After building the distance oracle, the algorithms can then answer the shortest-path query efficiently at the query answering stage. In this context, the runtime space requirement of the distance oracle is an important consideration to ensure scalability of algorithms to large, real-world graphs.

There have been a number of studies on how to accelerate the preprocessing and query answering time and minimize the runtime memory space. The seminal work by Thorup and Zwick in [10] coined the term "distance oracle" and proposed the lower bound of the space of a distance oracle based on Paul Erdös' Girth Conjecture [11]. Sommer et al. in [12] proved that for any high-girth (i.e., sparse) graphs, distance oracles with efficient query time and constant stretch (i.e., the worst-case accuracy for an approximate shortest-path query answering) require super-linear space. Akiba et al. [13] presented *PLL*, an exact shortest-path query algorithm based on the idea of 2-hop cover. PLL has outstanding performance on many real-world graphs despite not having been fully analyzed in theory [13]. Tree decomposition-based graph indexing algorithms (*TEDI*) [14]–[16] have also been developed. In fact, Raghavendra et al. in [17] showed that TEDI can significantly save the route query time and runtime space in real SDN systems. The main challenge of algorithms such as PLL and TEDI is that their preprocessing stages are extremely time-consuming for large-scale graphs (e.g., $O(n^2 \log n)$ [13], [16]), which presents a considerable performance bottleneck in practice.

In this paper, we present a new shortest-path query al-

[1]If implemented by a Fibonacci heap, Dijkstra's algorithm can run in $O(m + n \log n)$ time, where m is the number of edges in the graph.

978-1-5090-3281-5/16/$31.00 ©2016 IEEE

gorithm for weighted graphs that can construct the distance oracle at the preprocessing stage in a much lower time than existing algorithms. Our algorithm, called BBQ (*B*ottom-top-bottom distance oracle with *B*atch *Q*ueries), is also a kind of tree decomposition-based graph indexing algorithm. BBQ conducts the distance oracle computation by decomposing the graph into *bags*, for which local APSP calculations are performed. This approach reduces the preprocessing time for most real-world graphs. In addition, BBQ can leverage batch processing of multiple queries to significantly reduce the query answering time.

The specific contributions of our work are:

- Fast construction of a distance oracle: In BBQ, the distance oracle can be computed faster than in existing two-stage shortest-path query algorithms. Compared with $O(n^2 \log n)$ time for existing two-stage algorithms, BBQ requires only $O(nhk^2 \log k + |R|^2 \log |R|)$ time at the preprocessing stage, where $h \ll n$, $k \ll n$ and $|R| \ll n$ hold for real-world graphs.
- Batch query answering: BBQ removes the computation redundancies caused by tree path overlaps in multiple queries. To the best of our knowledge, our paper is the first work to optimize the running time of answering batches of shortest-path queries.
- Evaluation of BBQ on large-scale graphs: We present results from experimental measurements that show the performance of BBQ in comparison to single-stage shortest path algorithms (Dijkstra) and state-of-the-art two-stage algorithms (PLL and TEDI). Our results show that BBQ can amortize the preprocessing cost with as few as 145 queries (which is 3.1%–29.7% that of PLL and TEDI). If used in an SDN, BBQ can sustain up to 1,379,310 connection setup requests per second (compared to 500 requests per second for Dijkstra) and 3.5–33.3 times more network updates than PLL or TEDI.

The rest of this paper is organized as follows. In Section II, we introduce an overview of BBQ. Section III presents the construction of distance oracle in BBQ. In Section IV, we describe a query answering algorithm that optimizes the time complexity of answering batches of shortest-path queries. Section V presents the experimental results of BBQ for various real-world graphs and provides a comparison to Dijkstra, PLL, and TEDI. Related work is presented in Section VI. We conclude this paper in Section VII.

II. An Overview of BBQ

A computer network can be viewed as an undirected graph $G = (V, E)$, where $n = |V|$ and $m = |E|$. V is the set of nodes (e.g., SDN switches), and E is the set of links between nodes. In this paper, G is a weighted graph with a weight function $w : E \to \mathbb{R}^+$ which maps edges to positive valued weights. These weights represent the cost of using transmission links (e.g., delay, hop count, etc.). The weight of a path $\mathbf{p} =< v_1, v_2, \ldots, v_k >$ is defined as $w(\mathbf{p}) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$, which is the sum of all its constituent edges. Given $u, v \in V$, if there is a path between u and v, the shortest path of the two vertices is $\mathbf{p}$ with the minimum $w(\mathbf{p}) : u \xrightarrow{\mathbf{p}} v$. For convenience, we use the term *graph* to denote a *weighted undirected graph*.

A. Principles of BBQ

BBQ decomposes the graph G into a tree T_G, in which each tree node is a subset of V. The tree nodes of T_G are also called *bags*. Adjacent bags may have overlaps, i.e., a vertex in V can appear in more than one bags. For any vertex $u \in V$, all bags that contain u constitute a connected subtree. For building the distance oracle, we need to obtain the all-pairs shortest paths in each bag (i.e., "local APSP"). The most time-consuming step at the preprocessing stage is computing the local APSP results. Thus, we design a fast local APSP method following a bottom-top-bottom manner. Compared to the $O(n^2 \log n)$ time complexity in state-of-the-art approaches at their preprocessing stages [15], [16], BBQ runs in $O(nhk^2 \log k + |R|^2 \log |R|)$ time, where $h \ll n$, $k \ll n$ and $|R| \ll n$ hold for real-world graphs. Therefore, BBQ can significantly reduce the preprocessing time.

Moreover, BBQ can answer a batch of shortest-path queries at once. State-of-the-art shortest-path query algorithms concentrate on optimizing the time of answering a single shortest-path query. For obtaining the shortest path between $u, v \in V$, we need to traverse the tree path between the two bags $X(u)$ and $X(v)$, which are the bags containing u, v respectively. With multiple queries, the tree paths corresponding to those queries may have overlaps, which introduce redundant computations. To accelerate the speed of answering a batch of shortest-path queries, BBQ eliminates the computational redundancies by traversing T_G only once. Except for the $O(n)$ time of traversing T_G, compared to the $O(qk^2h)$ time complexity of state-of-the-art algorithms [14]–[16], BBQ only requires $O(qk \log k)$ $(k \ll n)$ time to answer q shortest-path queries.

B. Tree Decomposition

Definition 1 (Tree Decomposition as described in [18])**.** *A tree decomposition of a graph $G = (V, E)$, denoted as T_G, is a pair $\langle T, \chi \rangle$, where $T = \langle I, F \rangle$ is a tree, and $\chi = \{X_i | i \in I\}$ is a collection of subsets of V. T_G satisfies the following conditions:*

1. $\cup_{i \in I} X_i = V$*;*
2. *for each edge $(u, v) \in E$, there is $i \in I$ such that $u, v \in X_i$;*
3. *for each vertex $u \in V$, the set $\{p | u \in X_p\}$ induces a connected subtree.*

The decomposed tree T_G consists of a set of tree nodes that are the subsets of V. These tree nodes are also called *bags*. According to Condition 1, each vertex in V should appear in at least one bag. For each edge $e \in E$, the vertices of e should be in no less than one bag together (Condition 2). For any vertex $v \in V$, Condition 3 requires the connectivity of the induced subgraph of all bags containing v.

To distinguish the nodes in T_G and G, the nodes in T_G are called *bags*, and those in G are referred to as *vertices* in this paper. We denote the root of T_G as R.

The *treewidth* of T_G is $max_{i \in I}(|X_i| - 1)$. For simplicity, in our context, we denote the treewidth of T_G as $max_{i \in I}(|X_i|)$. Given a graph $G = (V, E)$, there are many kinds of T_G satisfying Definition 1. In the computer science community, researchers mainly concentrate on those T_G with small treewidths, though computing T_G with the smallest treewidth is an NP-hard problem [19].

TEDI [15] proposes a linear-time-complexity tree-decomposition method described in Algorithms 1 and 2. Notwithstanding that TEDI has a theoretical $O(n)$ treewidth for some extreme cases (e.g., complete graphs), we observe that the treewidth of TEDI is very small compared with n (i.e., $|R| \ll n$) for many real-world graphs [14], [15]. Since the real-world graphs are not extremely sparse, $k \ll n$ holds.

Algorithm 1 : Graph Reduction

Input: $G = (V, E)$, k
Output: bag stack S
1: initialize S
2: **for** i = 1 to k **do**
3: *remove_vertex(i)*
4: **if** G has no more than $k + 1$ vertices **then**
5: break
6: **end if**
7: **end for**
8: R = the residual vertices in G
9: push R into S
10: **return** S

11: **procedure** *remove_vertex(d)*
12: **while** TRUE **do**
13: **if** there is a vertex u with degree less than d **then**
14: $\{u_1, u_2, \ldots, u_d\}$ is the neighbors of u
15: build a clique for $C = \{u_1, u_2, \ldots, u_d\}$
16: push $\{u, u_1, u_2, \ldots, u_d\}$ into S
17: remove u and all its edges from G
18: **else**
19: break
20: **end if**
21: **end while**

In Algorithm 1, k is the reduction parameter that indicates we need to remove the vertices whose degrees are $\leq k$. Algorithm 1 removes the vertices from G until the degrees of residual vertices in G are all $\geq k$. Fig. 1 shows an example of Algorithm 1 with $k = 2$. Algorithm 1 begins with removing vertex 0 with 1-degree. After removing vertex 0 and its related edges, we build a bag $\{0, 4\}$ and push this bag into the *bag stack* S. At this time, we find that the degrees of residual vertices are all > 1, and then we move to reduce the vertices with 2-degree. We remove the vertices 1 and 2 in turn and push $\{1, 2, 3\}$ and $\{2, 3, 4\}$ into S. Let the remaining vertices

Algorithm 2 : Tree Decomposition

Input: bag stack S
Output: T_G
1: initialize T_G
2: R = pop up the top bag from S, is the root of T_G
3: **while** S is not empty **do**
4: pop up a bag $X_c = \{u, u_1, \ldots, u_d\}$ from S
5: find a bag X_f in T_G s.t. $X_f \cap X_c = \{u_1, \ldots, u_d\}$
6: add X_c into T_G as the child of X_f
7: **end while**
8: **return** T_G

$\{3, 4, 5\}$ be the root of T_G. After the graph reduction process, Algorithm 2 builds T_G shown in Fig. 2.

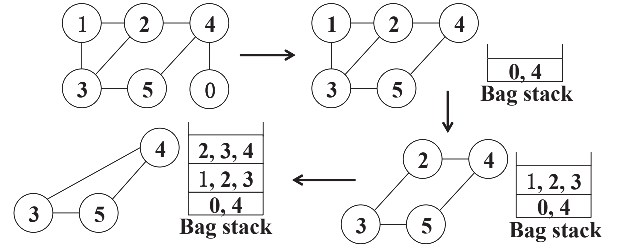


Fig. 1. An example of the graph reduction process in Algorithm 1.

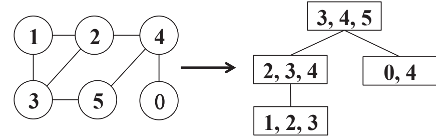


Fig. 2. A tree decomposition example with $k = 2$.

C. The Query Answering Stage

To answer a single shortest-path query (u, v), we first need to find the bags that contain u and v. Let $X(u)$ be the root of the induced tree, whose tree nodes all contain u. The *tree path* between $X(u)$ and $X(v)$ needs to be traversed for answering this query.

Definition 2. *A* tree path $\mathbf{tp}$ *is* $\mathbf{tp} = \{B_1, B_2, \ldots, B_l\}$*, where* $B_1, B_2, \ldots, B_l$ *are distinct bags in* T_G*, and* B_i*,* B_{i+1} *(*$i \in [l-1]$*) are neighbors.*

Let $\mathbf{tp}(X(u), X(v)) = \{B_1, B_2, \ldots, B_l\}$ be the tree path between $X(u)$ and $X(v)$, where $B_1 = X(u)$ and $B_l = X(v)$. For any given $u, v \in V$, there exists only one $\mathbf{tp}(X(u), X(v))$ due to the nature of tree. We denote $rdist(u, v)$ as the exact distance between u and v. Assuming that X_c and X_f are the adjacent bags on $\mathbf{tp}(X(u), X(v))$ in Fig. 3, we have [15], [16]

$$rdist(u, v) = min\{rdist(u, t) + rdist(t, v) | t \in X_c \cap X_f\}. \quad (1)$$

Hence, we can answer a single shortest-path query by a dynamic programming method [20] according to Eq. (1). The

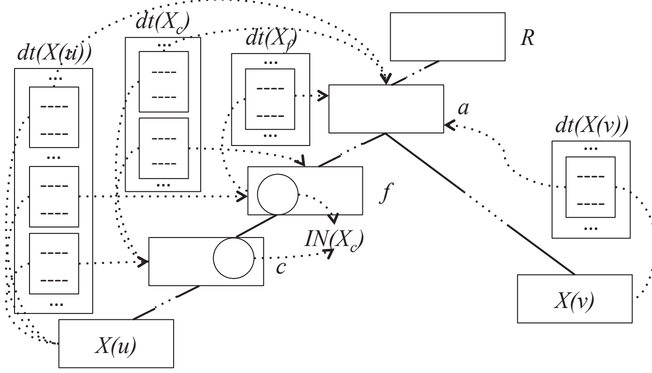


Fig. 3. The data structure of BBQ's distance oracle

time complexity of answering a single query is $O(k^2h)$, where k is the reduction parameter selected in Algorithm 1 and h is the height of T_G. In addition, Section IV presents that BBQ can optimize the answering time of batch queries in a different way.

D. The Preprocessing Stage

As shown in Algorithm 3, the construction of BBQ's distance oracle consists of four steps: (1) graph reduction; (2) tree decomposition; (3) the local APSP calculation; and (4) the ancestor distance tables calculation. The steps 1 (Algorithm 1) and 2 (Algorithm 2) help BBQ in building T_G. According to Eq. (1), the dynamic programming-based answering method requires pre-computing each bag's local APSP (i.e., the $rdist(\cdot,\cdot)$ value of any vertex pair in each bag) in step 3. Furthermore, we obtain the ancestor distance table for each non-root bag in step 4 to accelerate the answering of batch queries. Denote $IN(B)$ as $IN(B) = B \cap W$, where B is an arbitrary non-root bag and W is B's parent. Denote $U(X_e)$ as the set of X_e's all non-root ancestors. BBQ maintains an ancestor distance table $dt(X_e)$ for each non-root bag X_e. The table $dt(X_e)$ maps each item (a_i, b_j) to the value of $rdist(a_i, b_j)$, i.e., $dt(X_e)[(a_i, b_j)] = rdist(a_i, b_j)$, where $a_i \in IN(X_e)$ and $b_j \in IN(X_t)$ $(X_t \in U(X_e))$.

Algorithm 3 : Computing Distance Oracle

Input: $G = (V, E)$, k, the weight function w
Output: T_G, the local APSP results, dt
1: S = Graph Reduction($G(V, E)$, k)
2: T_G = Tree Decomposition(S)
3: $dist$ = Local All-Pairs Shortest Paths(G, S, w)
4: $dt, dist$ = Ancestor Distance Tables(T_G, $dist$)
5: **return** T_G, $dist$, dt

In summary, our goal at the preprocessing stage is to establish the distance oracle according to Algorithm 3. BBQ's distance oracle consists of T_G, the local APSP results $dist$ and the ancestor distance tables dt. At the query answering stage, BBQ answers a single shortest-path query via dynamic programming. In what follows, we show how BBQ can build the distance oracle quickly (Section III) and accelerate the answering time of batch shortest-path queries (Section IV).

III. Fast Building of a Distance Oracle

Algorithm 3 identifies four steps at the preprocessing stage: graph reduction, tree decomposition, local APSP calculation, and ancestor distance tables calculation. Since the first two steps both run in linear time (derived in Section II), we discuss the computational methods for the local APSP results $dist$ and the ancestor distance tables dt in this section.

A. The Local APSP Algorithm

A naïve solution to obtain all local APSP results is to utilize a classic APSP algorithm on G. Such a solution requires $O(n^2 \log n)$ time complexity [16]. However, due to the characteristic of tree decomposition, many pairs of vertices may not appear in the same bags, and thus it is not necessary to acquire the shortest paths between all vertex pairs. Therefore, we can design an efficient local APSP algorithm. We divide our fast local APSP algorithm into two parts, the *bottom-top* and the *top-bottom* processes shown in Algorithm 4.

Algorithm 4 : Local All-Pairs Shortest Paths

Input: G, bag stack S, weight function w
Output: distance function $dist$
1: $dist$ =*Bottom-Top Process(S, G, dist)*
2: $dist$ =*Top-Bottom Process(S, dist)*
3: **return** $dist$

1) The Bottom-Top Process: Recall from Section II that the order in which the bags come into the bag stack S indicates the generation sequence of bags. Algorithm 5 follows a bottom-top order to update $dist$, which is initialized with the weight function w.

Denote the ith $(i < sn)$ element from the bottom of S as $S_i = \{b_i, u_1^i, \ldots, u_l^i\}$, where b_i is the ith removed vertex according to Algorithm 1. Specifically, the top element S_{sn} in S is the root R of T_G. Theorem 1 shows that Algorithm 5 can partially compute the local APSP results.

Theorem 1. *Assume that the shortest path $\mathbf{p}(u, v)$ between u, v is $\mathbf{p}(u, v) = \langle p_1, p_2, \ldots, p_t \rangle$, where $p_1 = u$ and $p_t = v$. When finishing the loop body from line 7 to line 13 in Algorithm 5 for S_i, if $u, v \in S_i$ and $\{p_2, \ldots, p_{t-1}\} \subseteq \{b_1, b_2, \ldots, b_i\}$ hold, $dist^{(i)}(u, v)$ is the exact shortest-path distance between u, v.*

Proof. Denote the exact distance of the shortest path between $u, v \in V$ as $rdist(u, v)$. Consider a simple condition where $u, v \in S_i$ and the shortest path $\mathbf{p}(u, v)$ is $\langle u, b_i, v \rangle$ or $\langle u, v \rangle$. It is easy to obtain $dist^{(i)}(u, v) = rdist(u, v)$ after finishing the loop body from line 7 to line 13 in Algorithm 5. Next, we investigate a general case in which the number of hops of the shortest path is ≥ 2. W.l.o.g, we assume the shortest path $\mathbf{p}(u, v) = \langle p_1, p_2, \ldots, p_t \rangle$, where $\{p_2, \ldots, p_{t-1}\} \subseteq \{b_1, b_2, \ldots, b_j\}$. Denote ψ as $\psi = \{p_2, \ldots, p_{t-1}\}$ and an indexing function $lb(\cdot)$: $lb(x) = i$ $(x = b_i)$, $lb(x) = sn$

Algorithm 5 : Bottom-Top Process

Input: bag stack S, G, weight function w
Output: distance function $dist$

1: Initialize $dist^{(0)}$ with w
2: sn = the number of elements in S
3: **for** i = 1 **to** $sn-1$ **do**
4: $S_i = \{b_i, u_1^i, \ldots, u_l^i\}$
5: $dist^{(i)} = dist^{(i-1)}$
6: **for** j = 1 **to** l **do**
7: **for** k = 1 **to** l **do**
8: **if** $dist^{(i)}(u_j^i, u_k^i) > dist^{(i)}(u_j^i, b_i) + dist^{(i)}(b_i, u_k^i)$
9: $dist^{(i)}(u_j^i, u_k^i) = dist^{(i)}(u_j^i, b_i) + dist^{(i)}(b_i, u_k^i)$
10: **end if**
11: **end for**
12: **end for**
13: **end for**
14: **return** $dist^{(sn-1)}$

($x \in R$). According to the definition of the shortest path, we have $rdist(u,v) = \sum_{j=1}^{t-1} dist^{(0)}(p_j, p_{j+1})$. Therefore, we can calculate $rdist(u,v)$ by $dist^{(0)}$ and $\psi^{(0)} = \psi \cup \{u,v\}$. Suppose that p_x has the smaller $lb(p_x)$ than those of any other elements in ψ. After updating $dist^{(lb(p_x))}$ for vertices in $S_{lb(p_x)}$, we derive

$$\begin{aligned} rdist(p_{x-1}, p_{x+1}) &= dist^{(0)}(p_{x-1}, p_x) + dist^{(0)}(p_x, p_{x+1}) \\ &= dist^{(lb(p_x))}(p_{x-1}, p_{x+1}). \end{aligned}$$

Thus, we can remove p_x from ψ. We denote $\psi^{(1)} = \{p_1^{(1)}, p_2^{(1)}, \ldots, p_{t-1}^{(1)}\}$ as $\psi^{(0)} \setminus \{p_x\}$, and $rdist(u,v)$ can also be computed by $dist^{(lb(p_x))}$ and $\psi^{(1)}$. Then,

$$rdist(u,v) = \textstyle\sum_{j=1}^{t-2} dist^{(lb(p_x))}(p_j^{(1)}, p_{j+1}^{(1)}).$$

Let $ln(\cdot,\cdot)$ be an index-order function in which $ln(\psi,k)$ indicates the $lb(\cdot)$ value of the element with the kth smallest $lb(\cdot)$ value in ψ. Since we update $dist^{(ln(\psi,k))}$ for the vertices in all $S_{ln(\psi,k)}$ ($k \in [t-2]$), we can obtain

$$\begin{aligned} dist^{(ln(\psi,t-2))}(p_1^{(t-2)}, p_2^{(t-2)}) &= \sum_{j=1}^{2} dist^{(ln(\psi,t-3))}(p_j^{(t-3)}, p_{j+1}^{(t-3)}) \\ &= \sum_{j=1}^{t-2} dist^{(ln(\psi,1))}(p_j^{(1)}, p_{j+1}^{(1)}) \\ &= rdist(u,v). \end{aligned}$$

Therefore, this theorem is established. □

During the bottom-top process, the values in $dist^{(\cdot)}(\cdot,\cdot)$ are updated for each bag of S. The updating time complexity is $O(k^2)$ for each non-root bag. Consequently, the bottom-top process runs in $O(nk^2)$ time.

2) The Top-Bottom Process: After the bottom-top process, we partially compute the local APSP results in the bags of T_G according to Theorem 1. Then, we need a top-bottom process for computing the remaining local APSP results.

Algorithm 6 : Top-Bottom Process

Input: bag stack S, $dist$ returned from Algorithm 5
Output: distance function $dist$

1: sn = the number of elements in S
2: $dist$=*top_down(S, dist, sn)*
3: $dist$=*top_down(S, dist, sn* -1*)*
4: **return** $dist$

5: **procedure** *top_down(S, dist, st)*
6: $dist^{(st+1)} = dist$
7: **for** $i = st$ **downto** 1 **do**
8: G' = *build_subgraph(*S_i*,* $dist^{(i+1)}$*)*
9: $dist^{(i)} = dist^{(i+1)}$
10: Calculate the APSP for G' with the weight function $dist^{(i+1)}$
11: Update $dist^{(i)}(u,v)$ for all $u,v \in G'$ with the APSP results in line 10
12: **end for**
13: **return** $dist^{(1)}$

14: **procedure** *build_subgraph(*S_i*, dist)*
15: $V' = S_i$, $E' = \emptyset$
16: **for each** vertex pair (u,v) **in** S_i **do**
17: **if** $dist(u,v) < inf$ **then**
18: add edge (u,v) to E'
19: **end if**
20: **end for**
21: **return** $G'(V', E')$

Algorithm 6 describes the top-bottom process that completes the APSP algorithm in each bag from the top to the bottom of S.

Theorem 2. *When Algorithm 6 is finished, the local APSP results of all bags are complete.*

Proof. In a special case, the theorem is valid if a vertex pair $u,v \in R$, $dist^{(sn)}(u,v)$ is the exact shortest-path distance between u,v according to Theorem 1.

In the general scenario, where $u,v \in S_j$ and $\neg(u \in R \wedge v \in R)$, suppose the shortest path is $\mathbf{p}(u,v) =< p_1, p_2, \ldots, p_t >$, and thus $rdist(u,v) = \sum_{i=1}^{t-1} dist(p_i, p_{i+1})$, in which $dist$ is the input parameter. Since Theorem 1 holds, the shortest-path distance between u,v can be calculated by $\sum_{i=1}^{l-1} dist^{(j+1)}(cp_i, cp_{i+1})$, where $\mathbf{cp}(u,v) = \{cp_1, cp_2, \ldots, cp_l\} = \{p_i | p_i \notin \{b_1, \ldots, b_j\}\}$, $u = cp_1$, $v = cp_l$ and all elements of $\mathbf{cp}(u,v)$ are in $\mathbf{p}(u,v)$. There are two conditions as follows:

(i) **Condition 1**: $u = b_j$ or $v = b_j$

W.l.o.g, we assume that $u = b_j$. Denote ψ as $\psi = \{cp_2, \ldots, cp_{l-1}\}$. If $\psi \in R$, this theorem obviously is correct after finishing line 2 in Algorithm 6. Let nr be the number of ψ's elements that also belong to R and ns be the number of ψ's elements that are not in R. It is easy to see that $ns+nr = l-2$. Suppose that $cp_a, \ldots, cp_{a+b}$ are all in R. When executing line

3, we have

$$rdist(cp_{a-1}, cp_{a+b+1}) = \sum_{i=a-1}^{a+b} dist^{(sn)}(cp_i, cp_{i+1}). \quad (2)$$

Denote $\psi^{(0)}$ as $\psi^{(0)} = (\psi \backslash R) \cup \{u, v\} = \{cp_1^{(0)}, \ldots, cp_{ns+2}^{(0)}\}$, where $cp_1^{(0)} = u$ and $cp_{ns+2}^{(0)} = v$. Since Eq. (2) holds, we obtain $rdist(u, v) = \sum_{i=1}^{ns+1} dist^{(sn)}(cp_i^{(0)}, cp_{i+1}^{(0)})$.

We use the same indexing function lb(.) from Theorem 1. Suppose that $cp_x^{(0)}$ has the largest $lb(\cdot)$ value in $\psi^{(0)}$. Then,

$$rdist(cp_{x-1}^{(0)}, cp_{x+1}^{(0)}) = \sum_{i=x-1}^{x} dist^{(lb(cp_x^{(0)}))}(cp_i^{(0)}, cp_{i+1}^{(0)}).$$

Let $lm(\cdot, \cdot)$ be an index-order function in which $lm(\psi, i)$ indicates the $lb(\cdot)$ value of the element with the ith largest $lb(\cdot)$ in ψ. We define $\psi^{(n)} = \psi^{(n-1)} \setminus \{b_{lm(\psi,n)}\}$. We have

$$\begin{aligned} dist^{(lb(u))}(u, v) &= \sum_{i=1}^{1} dist^{(lm(\psi,ns))}(cp_i^{(ns)}, cp_{i+1}^{(ns)}) \\ &= \sum_{i=1}^{ns} dist^{(lm(\psi,1))}(cp_i^{(1)}, cp_{i+1}^{(1)}) \\ &= \sum_{i=1}^{ns+1} dist^{(sn)}(cp_i^{(0)}, cp_{i+1}^{(0)}) \\ &= rdist(u, v). \end{aligned}$$

Therefore, this theorem is true under this condition.

(ii) **Condition 2**: $u \neq b_j$ and $v \neq b_j$

In this condition, we can find a bag S_k $(k > j)$ that contains u, v, b_k $(b_k \in \psi)$. In S_k, $dist^{(k)}(u, v)$ can be updated by $dist^{(k)}(u, b_k) + dist^{(k)}(b_k, v)$. Since this proposition in Condition 1 holds and we run the APSP algorithm for S_k, there must be $dist^{(k)}(u, v) = rdist(u, v)$. Thus, the proposition is also correct in this condition.

Therefore, Theorem 2 holds. □

Although we carry out the APSP algorithm twice in each non-root bag for the conciseness of Theorem 2's proof, in fact, the APSP algorithm can be run only once for each bag. The time complexity of computing APSP is $O(nk^2 \log k)$ for all non-root bags and $O(|R|^2 \log |R|)$ for the root of T_G. Therefore, Algorithm 6 runs in $O(nk^2 \log k + |R|^2 \log |R|)$ time.

B. Ancestor Distance Tables

We maintain an ancestor distance table $dt(X_e)$ for each non-root bag X_e. BBQ uses Algorithm 7 to calculate the ancestor distance tables of all non-root bags.

As Fig. 3 shows, since Eq. (1) holds, if the values of $rdist(x, y)$ $(x \in IN(X(u)),\ y \in IN(X_c))$ have been computed, we can utilize an APSP algorithm on the induced graph G', containing the vertices in $IN(X(u))$, $IN(X_c)$ and $IN(X_f)$, to obtain $rdist(\cdot, \cdot)$ values for all vertex pairs in $IN(X(u)) \cup IN(X_f)$. Thus, the procedure *Update_Dt(·,·)* can compute the ancestor distance tables dt for all non-root bags in an iterative way.

Note that the values in dt are also stored in $dist$ according to Algorithm 7. That means that for any non-root bag X_e, if there is an item (a_i, b_j) $(a_i \in IN(X_e)$ and $b_j \in IN(X_t)$, where $X_t \in U(X_e))$ in $dt(X_e)$, we have $dist(a_i, b_j) = rdist(a_i, b_j)$. Therefore, $dist$ returned from Algorithm 3 contains the values of all local APSP results and the ancestor distance tables dt.

Algorithm 7 : Ancestor Distance Tables

Input: $dist$ returned from Algorithm 4, T_G
Output: $dist$, the ancestor distance tables dt

1: Initialize dt
2: R = the root of T_G
3: *Depth_First_Search(R)*
4: **return** dt, $dist$

5: **procedure** *Depth_First_Search(B)*
6: **if** B is a non-root bag **then**
7: *Update_Dt(B,B)*
8: $Item = \{(a_i, b_j) | a_i \in IN(B), b_j \in IN(X_t), X_t \in U(B)\}$
9: **for each** (x, y) in $Item$ **do**
10: $dt(B)[(x, y)] = dist(x, y)$
11: **end for**
12: **end if**
13: **for each** child C of B **do**
14: *Depth_First_Search(C)*
15: **end for**

16: **procedure** *Update_Dt(B,P)*
17: $PU = P$'s parent
18: **if** $PU \neq R$ **then**
19: Let G' be the induced graph of vertices in $IN(B)$, $IN(P)$ and $IN(PU)$
20: Compute the APSP for G' with the weight function $dist$
21: Update $dist$ according to line 20
22: $P = PU$
23: *Update_Dt(B,P)*
24: **end if**

C. Complexity

From the above analysis, the time complexity of computing the local APSP is $O(nk^2 \log k + |R|^2 \log |R|)$ and the local APSP algorithm spends $O(|R|^2 + nk^2)$ space to store the local APSP results. Algorithm 7 calculates the ancestor distance tables in $O(nhk^2 \log k)$ time, because each procedure *Update_Dt(·,·)* takes $O(hk^2 \log k)$ time and we run this procedure $O(n)$ times. Since each ancestor distance table takes $O(hk^2)$ space and each non-root bag has such a table, we need to store dt in $O(nhk^2)$ space. To sum up, the overall time complexity of building the distance oracle is $O(nhk^2 \log k + |R|^2 \log |R|)$ and the total space complexity is $O(nhk^2 + |R|^2)$.

IV. ANSWERING BATCH QUERIES

Section II shows that a single shortest-path query runs in $O(k^2 h)$ time, where k is the reduction parameter and h is the height of T_G. However, we also study the problem of

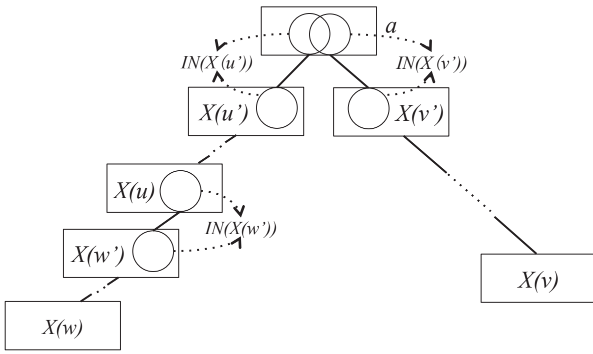


Fig. 4. An example of Algorithm 8

answering batch queries. Suppose that the number of shortest-path queries is q. A straightforward solution is to run the single query answering method q times, and thus the time complexity is $O(qk^2h)$. However, we can do better. In this section, we propose an $O(n+qk\log k)$ algorithm for answering q shortest-path queries.

A. Motivation

Suppose there are two queries (w, v) and (u, v) as Fig. 4. We first handle query (u, v). Recall from Eq. (1) that we find $rdist(u, v) = min\{rdist(u, t_1)+rdist(t_1, t_2)+rdist(t_2, v)\}$, where $t_1, t_2 \in L(X(u), X(v))$ and $L(X(u), X(v))$ is the *lowest common ancestor (LCA)* of $X(u)$ and $X(v)$. Of course, $X_a = L(X(u), X(v))$ in Fig. 4. To obtain $rdist(u, t_1)$ and $rdist(t_2, v)$, we need to traverse the tree paths, $\mathbf{tp}(X(u), X_a)$ and $\mathbf{tp}(X_a, X(v))$. When we start to answer the other query (w, v) as in Fig. 4, $\mathbf{tp}(X(w), X_a)$ and $\mathbf{tp}(X_a, X(v))$ need to be traversed. We find that $\mathbf{tp}(X(u), X_a)$ and $\mathbf{tp}(X_a, X(v))$ are traversed twice due to $L(X(u), X(v)) = L(X(w), X(v))$. With the number of queries increasing, the number of such traversal redundancies is sizable. Therefore, BBQ aims to remove the traversal redundancies for reducing the total time of answering batch queries.

B. Algorithm Description

Our answering algorithm for a batch of exact shortest-path queries is shown in Algorithm 8.

Fig. 4 shows an example of Algorithm 8. First, we use a classic LCA algorithm [21] (in line 1 of Algorithm 8) to help us in finding q LCAs within $O(n + q)$ time. For a query $(u, v) \in Q$, we suppose the LCA of $X(u)$ and $X(v)$ is X_a. Since $dt(X(u))$, $dt(X(v))$ have been pre-computed at the preprocessing stage, answering the shortest-path query (u, v) translates into finding the (u, v)-shortest path on the induced graph G' that contains the vertices in $\{u, v\}$, $IN(X(u))$, $IN(X(u'))$, $IN(X(v'))$ and $IN(X(v))$. It is worth noting that $IN(R) = \emptyset$.

C. Complexity

Except for the $O(n)$ time of traversing T_G, the classic algorithm [21] can answer a single LCA within $O(1)$ time [21]. For a query $(u, v) \in Q$, since $dt(X(u))$ and $dt(X(v))$ have been pre-computed at the preprocessing stage, we do not have to traverse $\mathbf{tp}(X(u), L(X(u), X(v)))$ and $\mathbf{tp}(L(X(u), X(v)), X(v))$ for solving Eq. (1). Eq. (1) can be solved by an $O(k \log k)$ SSSP algorithm on the induced graph G' that contains the vertices in $\{u, v\}$, $IN(X(u))$, $IN(X(u'))$, $IN(X(v'))$ and $IN(X(v))$, where the size of the vertex set of G' is $O(k)$. For each query $(u, v) \in Q$, we run an SSSP algorithm on an $O(k)$-scale graph. Therefore, the time complexity of Algorithm 8 is $O(n + qk \log k)$.

Algorithm 8 : Answering A Batch of Queries

Input: $T_G, dt, dist$, query set Q
Output: The shortest path query answer A
1: LS=*Find_LCA*(Q)
2: **for each** query (u, v) in Q **do**
3: X_a = $LS(X(u), X(v))$
4: $X(u')$ = a child of X_a and $X(u') \in U(X(u)) \cup \{X(u)\}$
5: $X(v')$ = a child of X_a and $X(v') \in U(X(v)) \cup \{X(v)\}$
6: G' is the induced graph of vertices in $\{u, v\}$, $IN(X(u))$, $IN(X(u'))$, $IN(X(v'))$ and $IN(X(v))$
7: Compute the shortest path between u, v on G' with the weight function $dist$
8: $A[(u, v)] = dist(u, v)$
9: **end for**
10: **return** A

TABLE I
DATASETS USED FOR PERFORMANCE EVALUATION

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	Network
Ark-IPv6	4,567	5,661	Autonomous Systems
SkitterAS	7,642	17,622	Autonomous Systems
Ark-IPv4	26,291	60,445	Autonomous Systems
GrQc	4,158	13,422	Collaboration
Wiki-Vote	7,066	100,736	Social
Gnutella	10,876	39,994	Internet P2P
Enron-3	33,696	18,0811	Email Communication
Slashdot	82,140	500,481	Social

V. EVALUATION

To show the effectiveness of BBQ, we compare its performance to that of a traditional single-stage algorithm (Dijkstra) and to that of state-of-the-art two-stage algorithms (PLL and TEDI). While asymptotic complexity provides some insights, it is also important to experimentally evaluate the performance of these algorithms (e.g., though PLL's theoretical complexity in the worst case is high, it performs well in practice).

A. Experimental Setup and Datasets

All experiments were run on a server with a single core of an Intel Xeon E5-2620 (2.4GHz) processor and 96 GB of main memory. All algorithms are implemented in C++ on Windows 10.

We use real-world graphs from the networking domain as well as from other domains for evaluation. All selected graphs are treated as undirected, weighted graphs. Table I lists

TABLE II
VALUE SELECTIONS OF k AND CORRESPONDING $|R|$ AND h

Dataset	k	$\|R\|$	h
Ark-IPv6	2	346	4
SkitterAS	8	503	7
Ark-IPv4	9	1,438	8
GrQc	8	785	13
Wiki-Vote	19	2,356	5
Gnutella	18	3,660	6
Enron-3	17	4,154	22
Slashdot	18	14,938	12

the graphs used in our experiments: (1) Ark-IPv6: an IPv6 Autonomous System (AS) network from [22]; (2) SkitterAS: an AS network from [23]; (3) Ark-IPv4: an IPv4 AS network from [24]; (4) GrQc: a graph created from Arxiv GR-QC (General Relativity and Quantum Cosmology) collaboration network [25] in which each vertex is a scientist and each edge represents a co-author relationship; (5) Wiki-Vote: a graph of the raw Wikipedia administrator election data [26], [27]; (6) Gnutella: a graph created from a snapshot of the Gnutella sharing network collected on Aug. 4, 2002 [28]; (7) Enron-3: an email communication network [29]; (8) Slashdot: a graph created from the Slashdot Zoo website on Feb. 21, 2009 [26] in which the vertices represent users and the edges correspond to friendships between users. Fig. 5 shows that the degree distributions of these real-world graphs obey a power law.

B. Preprocessing Time and Index Space

The time complexity of BBQ is determined by parameters k, $|R|$, and h. Since $|R|$, h are decided by parameter k, we first investigate how k impacts $|R|$ and h. Fig. 6 and Fig. 7 show that $|R|$ decreases and h grows with larger k. Therefore, we need to select k to leverage $|R|$ and h for obtaining the optimal preprocessing time. The values of our chosen k and the corresponding values for $|R|$ and h are shown in Table II.

Table III lists the preprocessing time (PT), the answering time for a single query (QT), and the index size (IS), which is the memory size of the distance oracle. The query time is averaged over 1,200,000 random queries. The results in Table III illustrate that PLL and TEDI take 3.5×–33.3× more time for preprocessing compared to BBQ.

C. Answering Time of Batch Queries

Although PLL, TEDI, and BBQ have similar answering time for single queries, as shown in Table III, BBQ can yield a significant performance improvement when answering batch shortest-path queries. Fig. 8 shows the log-log plot of the answering time for Ark-IPv6, SkitterAS and Ark-IPv4 for varying batch sizes. Because TEDI and PLL repeat single queries q times, where q is the batch size, their answering time linearly increases with batch size.

According to the analysis in Section IV, BBQ runs in $O(n + qk \log k)$ time for batch queries. When q is not large enough, the dominant factor of the running time is the first term, $O(n)$, and q has a minor contribution to the total time consumption. When q is large, q determines the total answering time. Fig. 8 shows the performance improvement of BBQ. When q is large, BBQ provides an orders-of-magnitude performance improvement over PLL and TEDI. (When q is not large enough, single queries can be used, which are marginally faster than PLL and TEDI as shown in Table III.) Fig. 8 also illustrates that the performance crossover (i.e., the batch size when the total time of BBQ's answering algorithm for batch queries is less than that of PLL and TEDI) is less than 10 for these AS networks.

D. Performance in SDN Context

To explore the performance of BBQ in the context of shortest-path finding on SDN controllers, we focus on the three AS networks, Ark-IPv6, SkitterAS and Ark-IPv4. State-of-the-art SDN systems, such as OpenDaylight [30] and ONOS [5], use Dijkstra to calculate the path of a connection [30].

We first present BBQ's ability to significantly increase the connection request rate on an SDN controller. Table IV shows the performance comparison for connection setup rates. If the distance oracle has been set up, we observe that BBQ can handle $21.3\times$ to $5,405.5\times$ more connections per second than Dijkstra and $1.1\times$ to $27.9\times$ more than PLL or TEDI.

If the network topology or link weights change, BBQ (as well as PLL and TEDI) needs to rebuild the distance oracle. Table V shows the maximum update rates. BBQ can sustain 3.5–33.3 times more network updates than PLL or TEDI.

Since Dijkstra does not use a distance oracle, we explore how many lookups are necessary to amortize the cost of this preprocessing. Table VI shows that after several hundreds of shortest-path queries, the total time of BBQ is less than that of Dijkstra. BBQ can amortize this cost more quickly than PLL or TEDI since its distance oracle construction time is much lower. Thus, BBQ can provide significant performance improvements for SDN controllers in large-scale, dynamic SDN.

VI. RELATED WORK

The classic shortest-path algorithms are Dijkstra, Bellman-Ford, Floyd-Warshall algorithms for weighted graphs and breath-first search for unweighted graphs [31]. However, one common assumption for these classic algorithms is that the whole graph is stored in main memory, which implies that both SSSP and APSP algorithms are not suitable for fast query answering on large-scale graphs.

In recent years, many shortest-path query algorithms with preprocessing stages have been proposed. One category of these algorithms is the *2-hop cover*-based algorithm. The authors in [32] proposed an exact distance oracle with the worst-case guarantee on the space and query answering complexity, and they considered a 2-hop cover method in which selects a subset L of vertices as landmark to facilitate the shortest-path query answering. An efficient method is *hierarchical hub labeling* [33], which achieves good results in road networks. An $O(\log n)$-approximation algorithm for minimizing the maximum label size of the 2-hop cover is presented by

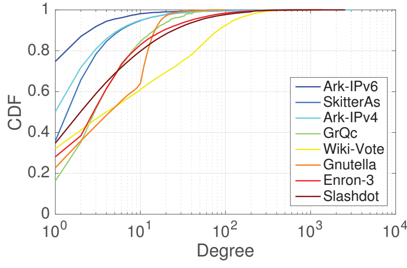


Fig. 5. Node degree distribution in graphs.

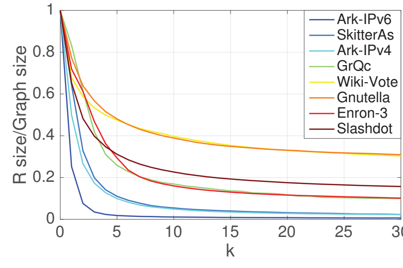


Fig. 6. Relationship between k and $|R|$.

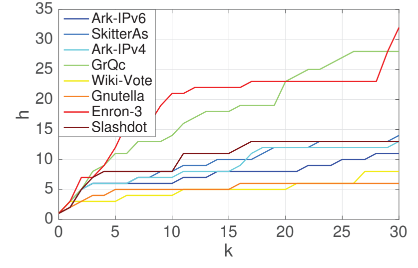
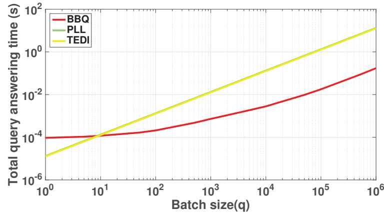


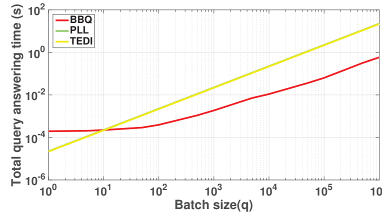
Fig. 7. Relationship between k and h.

TABLE III
PERFORMANCE COMPARISON BETWEEN BBQ AND OTHER SHORTEST-PATH ALGORITHMS

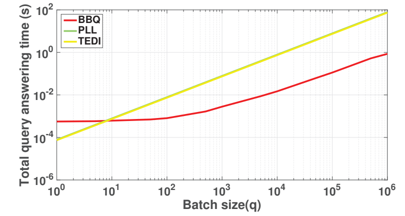
Dataset	Dijkstra	BBQ			PLL			TEDI		
	QT(μs)	PT(s)	QT(μs)	IS(MB)	PT(s)	QT(μs)	IS(MB)	PT(s)	QT(μs)	IS(MB)
Ark-IPv6	2,001	0.289	12.667	0.3	0.974	13.290	0.2	9.301	12.670	0.3
SkitterAS	3,752	1.231	21.521	1.6	5.025	22.490	0.5	29.551	21.668	0.8
Ark-IPv4	15,267	12.596	73.557	8.9	65.912	78.238	2.3	411.908	74.082	5.1
GrQc	2,135	1.140	12.400	2.1	5.860	12.943	1.6	9.490	12.397	1.4
Wiki-Vote	6,433	6.492	19.840	13.1	52.535	21.587	3.3	46.130	19.810	11.1
Gnutella	5,925	10.789	30.647	29.6	343.531	36.786	22.1	66.711	30.650	26.3
Enron-3	23,853	70.313	94.005	80.3	549.746	102.152	10.4	856.461	94.012	35.5
Slashdot	72,605	540.357	214.803	433.4	6,780.010	211.882	115.2	6,122.987	214.891	371.1



(a) Ark-IPv6: n=4.5 $\times 10^3$



(b) SkitterAS: n=7.6 $\times 10^3$



(c) Ark-IPv4: n=2.6 $\times 10^4$

Fig. 8. The answering time of batch shortest-path queries on Ark-IPv6, SkitterAS and Ark-IPv4

TABLE IV
CONNECTION SETUP RATE ON SDN CONTROLLER (q IS THE BATCH SIZE)

Dataset	Dijkstra	BBQ (q=1)	BBQ (q=10)	BBQ (q=50)	BBQ (q=100)	BBQ (q=500)	BBQ (q=1000)	PLL	TEDI
Ark-IPv6	500	78,945	84,746	303,030	478,469	1,054,852	1,379,310	75,245	78,927
SkitterAS	267	46,466	44,843	171,821	255,102	456,204	536,193	44,464	46,151
Ark-IPv4	66	13,595	16,207	71,429	123,762	301,205	356,761	12,782	13,499

TABLE V
TOPOLOGY UPDATE RATE ON SDN CONTROLLER

Dataset	BBQ	PLL	TEDI	vs. PLL	vs. TEDI
Ark-IPv6	3.584	1.027	0.108	3.5×	33.3×
SkitterAS	0.820	0.199	0.034	4.1×	24.2×
Ark-IPv4	0.080	0.015	0.002	5.3×	32.9×

TABLE VI
QUERIES NECESSARY TO AMORTIZE DISTANCE ORACLE CONSTRUCTION

Dataset	BBQ	PLL	TEDI	vs. PLL	vs. TEDI
Ark-IPv6	145	490	4678	29.7%	3.1%
SkitterAS	330	1347	7922	24.5%	4.2%
Ark-IPv4	825	4340	27112	19.1%	3.1%

[34]. A pruned landmark labeling algorithm was proposed in [13], which can handle queries on some large-scale graphs, such as social and website networks. The main challenge of 2-hop cover-based algorithms is to quickly find a small 2-hop

cover, as studied in [33].

Our work follows another direction of query algorithms, namely *tree-decomposition-based graph-indexing* algorithms. The seminal work in [15] describes a tree-decomposition-based graph index for the original graph at the preprocessing stage. This tree decomposition-based graph index facilitates shortest-path queries on graphs. Though finding a tree-decomposition graph index with the optimal size is an NP-hard problem, the tree-decomposition methods in [14], [15] can achieve good performance with both low time complexity and small index size for real-world graphs. The work in [14] improved the linear tree decomposition method from [15] by utilizing the core-fringe structure with a dense core and a tree-like fringe of real-world graphs. One of the biggest challenges of tree decomposition-based graph-indexing algorithms is preprocessing local APSP results in each bag.

VII. Summary and Conclusion

Our work addresses the important problem of finding the shortest path in a weighted graph. Efficient algorithms for this problem are important in networking and many other domains. A key challenge in tree-decomposition-based graph-indexing algorithms for shortest-path queries is the preprocessing run time of local APSP results in each node. State-of-the-art methods are extremely time-consuming for large graphs, which limits their scalability. Our BBQ algorithm uses a time-efficient preprocessing method and can accelerate the answering time through batch queries. Our experimental results show that BBQ outperforms existing algorithms both in terms of preprocessing time and answering time. The performance of SDN controllers (which need to process a shortest-path query for each connection) can be improved significantly with our algorithm.

Acknowledgments

The work was partially supported by STCSM under Grant No. 15DZ1100103, by 863 program under Grant No. 2015AA016106, by the National Science Foundation under Grant No. 1421448 and by the EU FP7 IRSES MobileCloud project under Grant No. 612212.

References

[1] S. Agarwal, M. Kodialam, and T.V. Lakshman. Traffic engineering in software defined networks. In *Proceedings of IEEE INFOCOM*, pages 2211–2219. IEEE, 2013.

[2] Y. Xi, L. Schwiebert, and W. Shi. Privacy preserving shortest path routing with an application to navigation. *Pervasive and Mobile Computing*, 13:142–149, 2014.

[3] A.V. Savkin and M. Hoy. Reactive and the shortest path navigation of a wheeled mobile robot in cluttered environments. *Robotica*, 31(02):323–330, 2013.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[6] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[7] R. Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 664–673. ACM, 2014.

[8] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–45, 2014.

[9] S. Chechik. Approximate distance oracles with constant query time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 654–663. ACM, 2014.

[10] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

[11] P. Erdös. Extremal problems in graph theory. In *Theory of Graphs and its Applications, Proceedings of the Symposium Held in Smolenice*. Citeseer, 1964.

[12] C. Sommer, E. Verbin, and W. Yu. Distance oracles for sparse graphs. In *Foundations of Computer Science, 50th Annual IEEE Symposium on*, pages 703–712. IEEE, 2009.

[13] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[14] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 144–155. ACM, 2012.

[15] F. Wei. Tedi: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 99–110. ACM, 2010.

[16] F. Wei-Kleiner. Tree decomposition-based indexing for efficient shortest path and nearest neighbors query answering on graphs. *Journal of Computer and System Sciences*, 82(1):23–44, 2016.

[17] R. Raghavendra, J. Lobo, and K.-W. Lee. Dynamic graph query primitives for sdn-based cloudnetwork management. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 97–102. ACM, 2012.

[18] N. Robertson and P.D. Seymour. Graph minors. ii. algorithmic aspects of treewidth. *Journal of algorithms*, 7(3):309–322, 1986.

[19] F.V. Fomin and Y. Villanger. Treewidth computation and extremal combinatorics. In *International Colloquium on Automata, Languages, and Programming*, pages 210–221. Springer, 2008.

[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[21] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[22] Caida. http://www.caida.org/data/active/ipv6_aslinks_dataset.xml.

[23] Caida. http://www.caida.org/tools/measurement/skitter/.

[24] Caida. http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml.

[25] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1):2, 2007.

[26] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1361–1370. ACM, 2010.

[27] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World Wide Web*, pages 641–650. ACM, 2010.

[28] R. Matei, A. Iamnitchi, and P. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.

[29] J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[30] OpenDaylight. https://www.opendaylight.org.

[31] D.E. Knuth. The art of computer programming, volume 1: fundamental algorithms, 1997.

[32] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[33] I. Abraham, D. Delling, A.V. Goldberg, and R.F. Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[34] M.A. Babenko, A.V. Goldberg, A. Gupta, and V. Nagarajan. Algorithms for hub label optimization. In *International Colloquium on Automata, Languages, and Programming*, pages 69–80. Springer, 2013.